Prof. Dr. Bernhard Nebel
Lehrstuhl für Grundlagen der Künstlichen Intelligenz

Albert-Ludwigs-Universität
Freiburg im Breisgau

Studienarbeit

Lösen rundenbasierter Erreichbarkeitsspiele unter Verwendung des AO*-Algorithmus und einer Relaxierungsheuristik

Autor: Pascal Bercher
Betreuer: Robert Mattmüller
Datum: 10. März 2008

Abstract

The purpose of this work is to solve a specific class of instances of 2-player-reachability-games and to represent all datastructures and intermediate steps in a consistent way. First, this class of instances is defined formally. In the next step, the AND/OR-graph is derived from this definition, which serves as search space for the search-algorithm. Afterwards, the algorithm AO* is introduced, which uses the heuristic of FF, an improvement of the HSP-heuristic, to perform the search.

Zusammenfassung

Ziel dieser Arbeit ist es, bestimmte Klassen von Instanzen eines 2-Spieler Erreichbarkeitsspiels zu lösen und alle hierbei verwendeten Datenstrukturen und Zwischenschritte in einer einheitlichen Notation anzugeben. Hierzu wird zunächst die behandelte Klasse an Problemen formal definiert. Im nächsten Schritt wird der UND/ODER-Graph aus dieser Definition abgeleitet, welcher als Suchraum für den eingesetzten Algorithmus dient. Schließlich wird der Algorithmus AO* vorgestellt, der zur Suche die Heuristik von FF nutzt, eine Verbesserung der HSP-Heuristik.

Einleitung

Diese Arbeit entstand aus dem AVACS-Projekt (Automatic Verification and Analysis of Complex Systems), Teilbereich S1. In diesem Teilbereich geht es im Wesentlichen um *System Verification*, also um das Nachweisen oder Widerlegen der Einhaltung bestimmter Spezifikationen formaler Systeme. Solche Systeme sind unter anderem Schaltkreise und Protokolle. In dieser Arbeit konzentrieren wir uns auf das Erzeugen von Gegenbeispielen. Auf die genauen Einzelheiten des Teilprojekts S1 und des Gesamtprojekts AVACS wird nicht weiter eingegangen, der interessierte Leser sei auf die AVACS-Website (http://www.avacs.org/) verwiesen, auf welcher sich genaue Beschreibungen finden lassen.
Der Beweis, dass das betrachtete System eine Spezifikation nicht umsetzt, besteht in dem betrachteten Ansatz aus der Suche nach einer Menge von Zielzuständen in dem Suchraum, der sich aus dem System ableiten lässt. Diese Aufgabe entspricht der Lösung eines Erreichbarkeitsspiels [5]. Um dieses Spiel zu lösen, wird eine Suche auf einem UND/ODER-Graphen [14] unternommen. Als Suchalgorithmus wird AO* [14] eingesetzt, der bei Expansion neuer Knoten auf eine Heuristik zurückgreift. Dabei wird die Heuristik des FF-Ansatzes [12, 13] eingesetzt, die eine Verbesserung der HSP-Heuristik [2] darstellt. Wird die Suche erfolgreich beendet, ist hiermit gezeigt, dass nicht alle Spezifikationen eingehalten werden. Weiterhin kann der gefundene Lösungsgraph vom Startzustand zu den gefundenen Zielzuständen verwendet werden, um den Fehler in der Umsetzung der Spezifikationen zu benennen.
Diese Arbeit stellt alle Schritte, die schließlich zur Lösung des Erreichbarkeitsspiels führen, in einer einheitlichen Notation dar. Weiterhin werden Modifikationen an AO* und der Heuristik vorgeschlagen, um die Sucheffizienz zu steigern.
Der zuvor bereits implementierte Ansatz [11] nutzte zur Suche nicht AO*, sondern Proof-Number Search [1, 15], die ebenso wie AO* bei Expansion neuer Knoten auf eine Heuristik zurückgreift. AO* ist besonders geeignet, wenn die Suche nach der Menge von Zielzuständen erfolgreich enden wird, da stets ein solcher Knoten expandiert wird, der wahrscheinlich (auf Grundlage der Heuristik) am schnellsten die Suche erfolgreich beendet. Falls die Suche nach den Zielzuständen nicht erfolgreich endet, wird AO* das Erreichbarkeitsspiel ebenfalls korrekt lösen, die Terminierungsgeschwindigkeit ist in diesem Falle aber deutlich geringer. Proof-Number Search ist hingegen nicht für den Fall optimiert, in welchem die Suche erfolgreich endet. Sie wird hauptsächlich ein-

gesetzt, wenn nicht abgeschätzt werden kann, ob die Suche erfolgreich oder erfolglos endet. Daher expandiert sie immer einen Knoten, der am schnellsten das Spiel löst, sei dies mit dem Ergebnis, dass Spieler 1, oder dass Spieler 2 eine Gewinnstrategie besitzt. Falls es also (wahrscheinlich) schneller möglich sein wird, die Suche erfolglos zu beenden, wird ein Knoten expandiert, der dies wahrscheinlich gewährleisten wird. Scheint es hingegen erfolgsversprechender, die Suche erfolgreich zu beenden, wird ein Knoten expandiert der die Suche wahrscheinlich erfolgreich beenden wird. Damit ist AO* besser für unsere Zwecke geeignet, da wir uns auf den Beweis der Nichteinhaltung der gegebenen Spezifikationen konzentrieren möchten und wir daher von der erfolgreichen Suche in dem Erreichbarkeitsspiel ausgehen.

Danksagungen

Ich möchte mich besonders bei meinem Betreuer, Robert Mattmüller, für die hervorragende Betreuung dieser Arbeit und die unzähligen Treffen bedanken, die mich bei dieser Arbeit sehr unterstützt haben.

Ein weiterer Dank gilt meinen Kommilitonen Christoph Betz und Marina Klingele für das Korrekturlesen dieser Arbeit und die vielen Verbesserungsvorschläge.

Inhaltsverzeichnis

1 Lösen rundenbasierter Erreichbarkeitsspiele

1.1 Die Problemstellung

Wir sind an der Lösung einer bestimmten Klasse von Problemen interessiert.
Diese von uns betrachtete Klasse von Problemen lässt sich am besten durch ein Erreichbarkeitsspiel [5] modellieren. Daher folgt zunächst die Defintion eines solchen Spiels.

Da das Hauptaugenmerk dieser Arbeit auf dem Lösen der für uns interessanten Probleme liegt, passen wir die Definition aus [5] von Erreichbarkeitsspielen unseren Bedürfnissen an. In dieser leichten Abwandlung wird statt einer Spielstruktur eine leicht angepasste Struktur verwendet, die weitestgehend mit der Definition einer rundenbasierten Spielstruktur übereinstimmt (vgl. Seite 6 in [5]). Der erhöhten Lesbarkeit zugute kommend, wird trotz geringer syntaktischer Unterschiede von einem Erreichbarkeitsspiel und von einer Spielstruktur gesprochen.

Definition 1 (Spielstruktur)**.**
Eine *Spielstruktur* $G = \langle S, MOVES, \Gamma_1, \Gamma_2, \delta, p \rangle$ besteht aus den folgenden Komponenten:

- Zwei Spielern 1 und 2 (diese sind implizit gegeben und tauchen damit nicht in der Definition auf). Wir nennen Spieler 1 auch ODER-Spieler und Spieler 2 UND-Spieler.
- Einer endlichen Menge S von Zuständen.
- Einer endlichen Menge $MOVES$ von Zügen.
- Zwei Zugzuweisungen $\Gamma_i : S \to 2^{MOVES}$, die jedem Spieler $i \in \{1, 2\}$ in jedem Zustand $s \in S$ die Menge der anwendbaren Züge zuweist. Wir fordern für alle $s \in S$, dass $\Gamma_i(s) = \emptyset$ für genau ein $i \in \{1, 2\}$. Wir definieren $\Gamma(s)$ als die Menge aller in einem Zustand s anwendbaren Züge. Es gilt $\Gamma(s) := \Gamma_1(s) \cup \Gamma_2(s)$.
- Einer partiellen Transitionsfunktion $\delta : S \times MOVES \to S$, die jedem Zustand $s \in S$ und jedem Zug $\gamma \in \Gamma(s)$ den Nachfolgezustand zuweist. Diese Abbildung ist partiell, da sie nur auf solchen Urbildern (s, γ) definiert ist, für die $\gamma \in \Gamma(s)$ für gegebenes $s \in S$.
- Einer Abbildung $p : S \to \{1, 2\}$ mit $p(s) = i$ genau dann, wenn $\Gamma_i(s) \neq \emptyset$ für $i \in \{1, 2\}$ und $s \in S$, die jedem Zustand den Spieler zuweist, der am Zug ist. Wir stellen an Γ_i für $i \in \{1, 2\}$ die Anforderung, dass $p(s) \neq p(s')$ für alle $s, s' \in S$ mit $s' = \delta(s, \gamma)$ für ein $\gamma \in \Gamma(s)$.

Definition 2 (Erreichbarkeitsspiel)**.**
Ein *Erreichbarkeitsspiel* $\mathcal{G}$ ist das Tripel $\langle G, R, s_0 \rangle$ mit einer Spielstruktur G, einem Startzustand $s_0 \in S$ und einer nichtleeren Menge von Zielzuständen $R \subseteq S$. Wir bezeichnen den Startzustand s_0 zusammen mit der Zielmenge R als *Aufgabe*.

Ziel des ODER-Spielers ist es, aus dem gegebenen Startzustand $s_0 \in S$ nach endlich vielen Zügen der beiden Spieler einen beliebigen Zielzustand $r \in R$ zu erreichen,

während der UND-Spieler dies zu verhindern versucht.
Es wird von optimaler Spielweise der beiden Spieler ausgegangen, d.h. der ODER-Spieler gewinnt genau dann, wenn er auf jeden Zug, den der UND-Spieler spielt, einen Zug entgegnen kann, der ihm nach endlich vielen Zügen beider Spieler das Erreichen eines beliebigen Zielzustandes garantiert. Dieses Gewinnkriterium wird in Definition 4 formalisiert.

Die Spielstruktur kann abstrakt als Transitionssystem angesehen werden, d.h. als Kodierung einer Miniwelt wie Schach oder eines konkreten Schaltkreises. Die Erweiterung der Spielstruktur zu einem Erreichbarkeitsspiel repräsentiert die Eigenschaft, die für dieses Transitionssystem nachzuweisen ist. Diese Eigenschaft kann durch Auswahl des Startzustands s_0 und der Zustände, welche in die Menge aller Zielzustände R übernommen werden, festgelegt werden.
Würde man als Beispiel Schach betrachten, so könnte man s_0 als die Grundstellung definieren und die Menge R als die Menge aller Schachstellungen, in denen Weiß gewinnt. Dann entspräche der Sieg des ODER-Spielers dem Finden einer optimalen Spielweise, die genau aussagt, in welcher Schachstellung welcher Zug zu spielen ist, um Schwarz letztendlich Matt zu setzen. In diesem Fall entspricht die zu zeigende Eigenschaft dem Lösen eines Spiels.
Eine andere Eigenschaft, die man zeigen kann, ist das Nichteinhalten von Spezifikationen offener Systeme: Möchte man zeigen, dass eine Umsetzung einer Menge von Spezifikationen Fehler aufweist, so kann man neben geeigneter Wahl des Startzustands s_0, R definieren als die Menge von Zuständen, die keinesfalls auftreten dürfen. Mit anderen Worten: Falls durch Eingaben aus der Umwelt bei optimalem Verhalten des Systems mindestens einer der Zustände aus R erreicht werden kann, ist mindestens eine Spezifikation verletzt und die Umsetzung damit fehlerhaft.

Unsere Definition eines Erreichbarkeitsspiels schreibt vor, dass genau ein Startzustand gegeben ist. Es mag aber durchaus interessant sein, ob die zu zeigende Eigenschaft auch für eine *Menge* von Startzuständen gilt. Sei also $\mathcal{G}' = \langle G, R, S_0 \rangle$ ein solches Erreichbarkeitsspiel, in dem G und R wie oben sind, aber $S_0 \subseteq S$ eine Menge von Startzuständen ist. Ein Erreichbarkeitsspiel $\mathcal{G}'$ gilt für den ODER-Spieler genau dann als gewonnen, wenn er für alle Anfangszustände $s_0 \in S_0$ einen Sieg für sich erzwingen kann.
Es kann leicht gezeigt werden, dass jedes Erreichbarkeitsspiel der Form $\mathcal{G}'$ auch als ein Erreichbarkeitsspiel der Form von $\mathcal{G}$ ausgedrückt werden kann, weswegen wir uns auf die bisherige Definition beschränken. Um dies zu bewerkstelligen, muss ein weiterer Zustand $s_0^* \notin S$ eingeführt werden, in dem der UND-Spieler am Zug ist und dessen Aktionen genau in die Zustände $s_0 \in S_0$ führen. Die Korrektheit dieser Vorgehensweise sollte nach der Erläuterung des Suchalgorithmus auf dem UND/ODER-Graphen ersichtlich werden (siehe Kapitel 1.3: Suche im UND/ODER-Graphen).

Definition 3 (Strategie)**.**
Eine *Sequenz* σ ist ein Tupel $(s_0, \ldots, s_n)$ mit $s_j \in S$ für $j \in \{0, \ldots, n\}$ und $n \in \mathbb{N} \cup \{\infty\}$. Wir nennen σ *endliche Sequenz*, falls $n \in \mathbb{N}$ und *unendliche Sequenz*, falls $n = \infty$. Es sei $s \in S$ und $\sigma = (s_0, \ldots, s_n)$ eine endliche Sequenz. Dann ist σs die Sequenz $(s_0, \ldots, s_n, s)$. Die Menge aller endlichen, nichtleeren Sequenzen bezeichnen wir mit S^+ und die Menge aller unendlichen Sequenzen mit S^ω.

Eine *mögliche Historie* ist eine Sequenz von Zuständen, die im zugrunde liegenden Erreichbarkeitsspiel $\mathcal{G}$ tatsächlich auftreten könnte, d.h. eine mögliche Historie σ ist eine Sequenz $(s_0, \ldots, s_n)$ mit:

- s_0 ist Startzustand des Erreichbarkeitsspiels.
- $s_{j+1} := \delta(s_j, \gamma)$ für ein $\gamma \in \Gamma(s_j)$ und für alle $0 \leq j < n$.

Wir schreiben fortan kurz *Historie*, statt *mögliche Historie*.

Es sei S_i^+ die Menge aller endlichen Historien, an deren Ende Spieler i am Zug ist, also gilt $S_i^+ := \{ \sigma = (s_0, \ldots, s_n) \in S^+ \mid \sigma \text{ ist endliche Historie und } p(s_n) = i \}$.

Eine *Strategie* π_i für Spieler $i \in \{1, 2\}$ ist eine Abbildung $S_i^+ \to MOVES$, die jeder Historie $\sigma = (s_0, \ldots, s_n) \in S_i^+$ genau einen möglichen Zug $\gamma \in \Gamma_i(s_n)$ zuweist.

Eine *gedächtnislose Strategie* für Spieler $i \in \{1, 2\}$ ist eine Strategie, so dass für alle Historien $\sigma s, \tau s \in S_i^+$ gilt: $\pi_i(\sigma s) = \pi_i(\tau s)$. Eine gedächtnislose Strategie ist damit unabhängig von Historien, da der selektierte Zug für alle Historien, die zum Zustand s führen, stets der gleiche ist, womit der selektierte Zug ausschließlich vom aktuellen Zustand abhängt.

Da gedächtnislose Strategien nur vom letzten Zustand der Historie abhängen, schreiben wir für $i \in \{1, 2\}$ und eine Historie $\sigma s \in S_i^+$ statt $\pi_i(\sigma s)$ kurz $\pi_i(s)$.

Definition 4 (Ziel des ODER-Spielers)**.**
Dank der Definition einer Strategie sind wir nun in der Lage, das *Ziel des ODER-Spielers* genau zu definieren, d.h. wir werden ein formales Kriterium angeben, wann der ODER-Spieler das Erreichbarkeitsspiel gewinnt, und wann der UND-Spieler.

Da wir fordern, dass es für jedes $s \in S$ *genau* ein $i \in \{1, 2\}$ mit $\Gamma_i(s) = \emptyset$ gibt und immer derjenige Spieler i ziehen muss, dessen Zugzuweisungsmenge Γ_i nicht leer ist, folgt, dass jede durch zwei beliebige Strategien induzierte Historie unendlich ist. Wir bezeichnen eine solche durch zwei gedächtnislose Strategien π_1 und π_2 induzierte Historie als Outcome O und die Menge aller Zustände in dieser Historie als *States*.

$$O(\pi_1, \pi_2) = (s_0, s_1, s_2, \ldots) \text{ mit } s_{j+1} := \delta(s_j, \pi_{p(s_j)}(s_j)) \text{ für } j \in \mathbb{N} \cup \{0\}$$
$$\mathit{States}(O(\pi_1, \pi_2)) = \{s_0, s_1, s_2, \ldots\} \text{ mit } s_{j+1} := \delta(s_j, \pi_{p(s_j)}(s_j)) \text{ für } j \in \mathbb{N} \cup \{0\}$$

Die Strategie π_1 gewinnt gegen Strategie π_2 gdw. $\mathit{States}(O(\pi_1, \pi_2)) \cap R \neq \emptyset$.

Der ODER-Spieler gewinnt das Erreichbarkeitsspiel $\mathcal{G}$ genau dann, wenn er eine gedächtnislose Strategie π_1 besitzt, die gegen alle gedächtnislosen Strategien π_2 des UND-Spielers gewinnt. Eine solche Strategie nennen wir *Gewinnstrategie*.

Es bleibt zu zeigen, dass diese Definition einer Gewinnstrategie zulässig ist, d.h. dass das Erreichbarkeitsspiel $\mathcal{G}$ im Sinne der Aufgabenstellung (vgl. Definition Erreichbarkeitsspiel) tatsächlich für den ODER-Spieler als gewonnen angesehen werden darf, obwohl wir uns bei der Definition einer Gewinnstrategie auf gedächtnislose Strategien beschränken. Es gilt jedoch, dass in jedem Parity-Spiel beide Spieler gedächtnislos gewinnen (Theorem 2.14 aus [7]).
Da sich jedes Erreichbarkeitsspiel als Parity-Spiel ausdrücken lässt, ist gezeigt, dass die Definition der Gewinnstrategie hinreichend ist, um die Gewinnbedingung des ODER-Spielers zu formalisieren.

1.2 Abstraktion

Sei nun eine Instanz eines solchen Erreichbarkeitsspiels gegeben. Um dieses Spiel zu lösen, übersetzen wir das Erreichbarkeitsspiel in einen UND/ODER-Graphen [14], in welchem sich UND-Knoten und ODER-Knoten stets abwechseln, was den Sachverhalt wiederspiegelt, dass der UND- und der ODER-Spieler abwechselnd ziehen. Wir haben den Sieg des ODER-Spielers formal darüber definiert, dass er eine Gewinnstrategie finden muss. Intuitiv muss der ODER-Spieler auf jeden Zug, den der UND-Spieler spielt,

einen Zug entgegnen können, der ihm nach endlich vielen Zügen beider Spieler das Erreichen eines beliebigen Zielzustandes garantiert. Damit ist es hinreichend, wenn in jedem Zustand *mindestens einer* seiner möglichen Züge ein solcher ist, der ihm irgendwann den Sieg garantiert. Dies entspricht einem logischen ODER, weswegen wir Zustände, in denen der ODER-Spieler am Zug ist, mit ODER-Knoten identifizieren. Für diesen gewählten Zug muss aber gewährleistet sein, dass *jeder* Zug des UND-Spielers im daraus resultierenden Zustand dem ODER-Spieler irgendwann den Sieg garantiert. Damit entsprechen Zustände, in denen der UND-Spieler am Zug ist, den UND-Knoten.

An dieser Stelle sei auf die Konvention aufmerksam gemacht, dass wir immer genau dann von *Zuständen* sprechen, wenn von der eigentlichen Problemstellung, also dem Erreichbarkeitsspiel die Rede ist, während wir von *Knoten* sprechen, falls der UND/ODER-Graph gemeint ist, der im Folgenden eingeführt wird. Analog identifizieren wir Züge mit Kanten und verwenden den entsprechenden Begriff gemäß Kontext. Diese Unterscheidung spiegelt lediglich den Kontext wieder, sonstige Unterschiede existieren nicht. Wir betrachten einen Spezialfall der Definition eines UND/ODER-Graphen. Erneut sprechen wir trotz der Behandlung eines Spezialfalles von einem UND/ODER-Graphen.

Definition 5 (UND/ODER-Graph)**.**
Ein *UND/ODER-Graph* $\mathcal{G}^{A/O}$ ist ein Tripel $\langle V_{AND}, V_{OR}, E \rangle$, bestehend aus:

- Einer endlichen Menge von UND-Knoten V_{AND}.
- Einer endlichen Menge von ODER-Knoten V_{OR}.
- Einer Menge $E \subseteq (V_{AND} \times V_{OR}) \cup (V_{OR} \times V_{AND})$ von Kanten.
  - Eine Kante $(a, b) \in E$ bezeichnen wir als UND-Kante, falls $a \in V_{AND}$ und entsprechend als ODER-Kante, falls $a \in V_{OR}$.
  - Falls $(a, b) \in E$, so bezeichnen wir a sowohl als Vaterknoten von b, als auch als Vaterknoten von (a, b). b ist entsprechend der Kind-Knoten von a und von (a, b).

V_{AND} und V_{OR} seien disjunkt, d.h. es gelte $V_{AND} \cap V_{OR} = \emptyset$.

Definition 6 (Implizite Instanz des UND/ODER-Graphen)**.**
Die *implizite Instanz des UND/ODER-Graphen* entspricht der vollständigen Übersetzung des Erreichbarkeitsspiels in den UND/ODER-Graphen:

$$
\begin{aligned}
V_{OR} &= \{\ s \in S \mid p(s) = 1\ \} \\
V_{AND} &= \{\ s \in S \mid p(s) = 2\ \} \\
E &= \{\ (s, s') \mid s \in V_{OR},\ s' \in V_{AND} \text{ und es ex. ein } \gamma \in \Gamma_1(s) \text{ mit } \delta(s, \gamma) = s'\} \\
&\quad \cup \{\ (s, s') \mid s \in V_{AND},\ s' \in V_{OR} \text{ und es ex. ein } \gamma \in \Gamma_2(s) \text{ mit } \delta(s, \gamma) = s'\}
\end{aligned}
$$

Die vollständige Übersetzung des Erreichbarkeitsspiels in den UND/ODER-Graphen ist trotz endlicher Größe (da V_{OR} und V_{AND} endlich) im Allgemeinen viel zu groß, um sie explizit zu repräsentieren. Der vollständige Graph ist daher immer nur implizit in Form der obigen Definition gegeben.
Wollte man den vollständigen Graphen dennoch explizit berechnen, so könnte man dies bewerkstelligen, indem man zunächst s_0 als einzigen ODER-Knoten annimmt, anschließend mit Hilfe der Zugzuweisungen und der Transitionsfunktion alle Nachfolgezustände bestimmt und hierdurch zusätzliche Kanten sowie die ersten UND-Knoten gewinnt. Diese Vorgehensweise wird rekursiv fortgesetzt, bis der gesamte Graph expandiert wurde.
Auf eine algorithmische Beschreibung wird verzichtet, da es nicht in unserem Interesse ist, den gesamten UND/ODER-Graphen zu berechnen.

Definition 7 (Explizite Instanz des UND/ODER-Graphen)**.**
Wir betrachten stets einen Teilgraphen des impliziten UND/ODER-Graphen, in welchem nur die Knoten expandiert werden, die für den Algorithmus gerade notwendig sind. Diese *explizite Instanz des UND/ODER-Graphen* wird im Folgenden auch als *expliziter Graph* bezeichnet und abgekürzt durch $\mathcal{G}'^{A/O}$.

1.3 Suche im UND/ODER-Graphen

Wir wissen nun, wie wir die gegebene Instanz des Erreichbarkeitsspiels in eine Instanz des UND/ODER-Graphen übersetzen. Im nächsten Schritt führen wir die Suche auf diesem neuen Suchraum aus, die den expliziten UND/ODER-Graphen stückweise expandiert, bis entweder eine Lösung gefunden oder der Graph vollständig expandiert wurde, ohne dass eine Lösung gefunden werden konnte.

Wir suchen auf dem UND/ODER-Graphen unter Verwendung des AO*-Algorithmus. AO* ist ein informierter Algorithmus, der ähnlich wie A* [9, 10] die Kosten der Kanten verwendet, sowie eine Heuristik h zur Schätzung der Distanz zu einem Zielknoten. Da jede Kante des Graphen einer Aktion des Erreichbarkeitsspiels entspricht und weiterhin alle Aktionen dieselben Kosten besitzen, werden allen Kanten identische Kosten (gleich 1) zugewiesen. Auf die verwendete Heuristik wird in Kapitel 1.4 näher eingegangen, wir setzen zunächst lediglich die Existenz voraus, wobei $h(n) = 0$ gilt für alle $n \in R$. AO* berücksichtigt weiterhin die spezielle Eigenschaft eines UND/ODER-Graphen, dass nur mindestens eine ausgehende Kante eines ODER-Knotens in einen Zielknoten oder einen bereits als gewonnen markierten Knoten münden muss, während dies bei einem UND-Knoten für alle ausgehenden Kanten erforderlich ist.

Im Folgenden werden die Begriffe Heuristik und Kostenschätzung strikt unterschieden, wobei die zugrundeliegende Bedeutung beider Begriffe stark verwandt ist. Mit Heuristik wird der Wert bezeichnet, der durch unsere Heuristikfunktion, hier also die FF-Heuristik (siehe dazu Kapitel 1.4: Die Heuristik) errechnet wird. Die Heursitik wird für einen Knoten genau dann berechnet, wenn dieser das erste Mal in den Suchgraphen aufgenommen wird. Wird ein solcher Knoten später expandiert, kann diese Heuristik unter Einbeziehung der Heuristiken der Kindknoten verbessert werden. Als Kostenschätzung bezeichnen wir die für einen Knoten augenblicklich vorliegende Schätzung der Kosten, die notwendig sind, um den aktuellen Knoten zu lösen. Diese Kosten entsprechen initial der Heuristik, im Allgemeinen sind sie aufgrund von Aktualisierungen aber besser als diese. Der Hauptunterschied zwischen der Heuristik und der Kostenschätzung liegt darin, dass die Heuristik lediglich die Distanz zum nahegelegensten Zielzustand errechnet, während die Kostenschätzung die Struktur des UND/ODER-Graphen berücksichtigt.
Wir werden auch dann von Kostenschätzung sprechen, wenn es keine Rolle spielt, ob von der Heuristik oder der aktualisierten Heuristik die Rede ist, da die Heuristik einer initialen Kostenschätzung entspricht. Eine Unterscheidung der Begriffe Kostenschätzung und Heuristik wird immer nur dann von Bedeutung sein, wenn speziell auf die Heuristik Bezug genommen wird.

Definition 8 (Lösungsgraph)**.**
Ein *Lösungsgraph* eines impliziten Graphen $\mathcal{G}^{A/O}$ ist ein zusammenhängender Teilgraph von $\mathcal{G}^{A/O}$, dessen Wurzelknoten s_0 ist und dessen ODER-Knoten nur eine ODER-Kante besitzen, während alle UND-Kanten der UND-Knoten erhalten bleiben, so dass der resultierende Graph zyklenfrei ist und alle Blätter (Knoten ohne Kinder) Zuständen aus R entsprechen.

Knoten, die Zuständen aus R entsprechen, bezeichnen wir als gelöst.
Die Tatsache, dass ein Lösungsgraph für jeden ODER-Knoten nur eine ODER-Kante enthält, während alle UND-Knoten alle UND-Kanten enthalten, rührt aus der Lösungsbedingung des Erreichbarkeitsspiels her, dass der ODER-Spieler genau dann gewinnt, wenn er auf *jeden* Zug des UND-Spielers *einen* Zug entgegnen kann, der ihm nach endlich vielen Zügen beider Spieler das Erreichen eines beliebigen Zielzustandes garantiert.
Die Tatsache, dass der Lösungsgraph im Gegensatz zum impliziten Graphen $\mathcal{G}^{A/O}$ stets zyklenfrei ist, folgt nach Konstruktion.

Aus dem Lösungsgraphen kann direkt die gesuchte Gewinnstrategie des ODER-Spielers abgelesen werden, indem für jeden ODER-Knoten s und jede dazugehörige ODER-Kante (die einem Zug γ des ODER-Spielers entspricht), die Funktionsvorschrift $s \mapsto \gamma$ zur Abbildung π_1 hinzugefügt wird.

Definition 9 (Optimaler Lösungsgraph)**.**
Ein *optimaler Lösungsgraph* entspricht einem Lösungsgraphen, der eine minimale Anzahl von Kanten besitzt.

Definition 10 (Optimaler Teillösungsgraph)**.**
Zunächst ist festzuhalten, dass der Begriff *optimaler Teillösungsgraph* irreführend sein kann. Es handelt sich hier nicht um einen Teilgraphen des optimalen Lösungsgraphen, sondern um einen solchen, der auf Grundlage der Kostenschätzungen am wahrscheinlichsten zu einem solchen erweitert werden kann. Dabei ist es sogar möglich, dass ein optimaler Teillösungsgraph zum gesuchten Lösungsgraphen mit Ausnahme des Startknotens s_0 disjunkt ist.

Ein *Optimaler Teillösungsgraph* ${\mathcal{G}^*}^{A/O}$ eines expliziten Graphen ${\mathcal{G}'}^{A/O}$ ist ein zusammenhängender Teilgraph von ${\mathcal{G}'}^{A/O}$, dessen Wurzelknoten s_0 ist und dessen ODER-Knoten genau eine ODER-Kante haben, deren Kindknoten eine Kostenschätzung haben, die kleinergleich allen Kostenschätzungen der anderen Kindknoten ist. Eine ODER-Kante, die diese Eigenschaft erfüllt, bezeichnen wir als markiert. Falls mehrere Kanten desselben ODER-Knotens diese Eigenschaft erfüllen, wird eine beliebige markiert.
Damit enspricht ein optimaler Teillösungsgraph einem Teilgraphen von ${\mathcal{G}'}^{A/O}$, der auf Grundlage der Kostenschätzungen am vielversprechendsten zu einem Lösungsgraphen ausgebaut werden kann, da im korrespondierenden Erreichbarkeitsspiel der ODER-Spieler nur solche Züge spielt, die ihn am schnellsten zu einer Lösung führen.

1.3.1 Der Suchalgorithmus AO*

Betrachten wir vor der algorithmischen Beschreibung von AO* zunächst seine Vorgehensweise. Im Wesentlichen gibt es zwei Schritte:

In einem Vorwärtsschritt wird von allen noch unexpandierten Knoten ein Knoten expandiert, der die beste Heuristik besitzt und sich zusätzlich in einem optimalen Teillösungsgraphen von ${\mathcal{G}'}^{A/O}$ befindet. Expansion ist der Vorgang des Erzeugens aller Nachfolger des zu expandierenden Knotens und des Einfügens in ${\mathcal{G}'}^{A/O}$. Sofort bei Erzeugung eines neuen Knotens wird für diesen ein Heuristikwert errechnet. Ist einer der erzeugten Knoten ein Terminalknoten (d.h. ein Zielknoten $r \in R$), so wird dieser als gelöst markiert.

In einem Rückwärtsschritt werden nun diese neuen Heuristikwerte verwendet, um die bisherigen Kostenschätzungen anzupassen. In diesem Hochpropagierungsschritt wird ebenfalls dafür gesorgt, dass Knoten korrekt als gelöst markiert werden. Ein Vaterknoten gilt als gelöst, falls er ein UND-Knoten ist und alle Kindknoten gelöst sind oder

falls er ein ODER-Knoten ist und mindestens ein Kindknoten gelöst ist.

AO* terminiert stets nach endlich vielen Schritten. Sobald AO* terminiert ist, wurde entweder eine Lösung (d.h. eine Gewinnstrategie für den ODER-Spieler) gefunden, oder es existiert keine solche Lösung. Mit anderen Worten: AO* ist vollständig und korrekt.

Es folgt eine kurze Klärung der im Algorithmus verwendeten Terminologie:

- $c(n)$ bezeichne die aktuellen geschätzten Kosten des Knotens n, die notwendig sind, um diesen zu lösen.
- $h(n)$ bezeichne den Wert der Heuristik des Knotens n.
- $\mathit{SOLVED}(n) \in \{\mathit{true}, \mathit{false}\}$ bezeichne, ob der Knoten n gelöst ist oder nicht. Wir bezeichnen den Wert von $\mathit{SOLVED}(n)$ als Lösungsstatus des Knoten n.
- $\mathit{MARKED}(e) \in \{\mathit{true}, \mathit{false}\}$ bezeichne, ob die Kante e markiert ist oder nicht.
- $\mathit{depth}(n)$ bezeichne für einen Knoten n seine initiale Tiefe. Aufgrund von Transpositionen (mehrere Pfade, die zu demselben Knoten führen) und Zyklen ist für einen Knoten nicht in eindeutiger Weise definiert, wie dessen Tiefe zu berechnen ist. Um Rechenaufwand zu sparen, wird $\mathit{depth}(n)$ einmal berechnet und danach nicht mehr aktualisiert. Damit ist $\mathit{depth}(n)$ eine obere Schranke für die Distanz vom Wurzelknoten s_0 zum Knoten n in $\mathcal{G}^{\mathcal{A}/\mathcal{O}}$.

Algorithmus : AO*

input : Erreichbarkeitsspiel $\mathcal{G} = \langle\langle S, MOVES, \Gamma_1, \Gamma_2, \delta, p\rangle, R, s_0\rangle$ mit dem impliziten Graphen $\mathcal{G}^{A/O} = \langle V_{AND}, V_{OR}, E\rangle$

output : Expliziter Graph $\mathcal{G'}^{A/O} = \langle V'_{AND}, V'_{OR}, E'\rangle$ und (falls existent) ein Lösungsgraph $\mathcal{G}^{*A/O} := \langle V^*_{AND}, V^*_{OR}, E^*\rangle \subseteq \mathcal{G'}^{A/O}$

`/* Erstelle expliziten Suchgraphen` $\mathcal{G'}^{A/O}$`, der initial nur aus dem Startknoten besteht. Errechne initiale Werte des Startknotens. */`

$\mathcal{G'}^{A/O} := \langle \emptyset, \{s_0\}, \emptyset\rangle$;

$c(s_0) := h(s_0)$;

$depth(s_0) := 0$;

if $s_0 \in R$ **then** $SOLVED(s_0) := true$ **else** $SOLVED(s_0) := false$;

while $SOLVED(s_0) \neq true$ *und* $\mathcal{G'}^{A/O} \neq \mathcal{G}^{A/O}$ **do**

  partialSolution();

  `/* Selektiere aus` $\mathcal{G}^{*A/O}$ `einen noch nicht expandierten und noch nicht gelösten Knoten` n `mit kleinster Heuristik */`

  Sei $fringe := \{\ n \in V^*_{AND} \cup V^*_{OR} \mid SOLVED(n) = false$ und es ex. kein $n' \in V'_{AND} \cup V'_{OR}$ mit $(n, n') \in E'\}$;

  Sei $n = \underset{n'' \in fringe}{\text{argmin}}\ h(n'')$;

  expand(n);

  `/* Erstelle die Updatemenge` U `und die Kontrollmenge` C`. */`

  $U = \{n\}$; $C = \{n\}$;

  while $U \neq \emptyset$ **do**

    $m :=$ *updateU*();

    updateDistanceAndSolved(m);

    selectParents(m);

  end

end

Algorithmus 1 : AO*

Nun folgt eine Auflistung der Implementierung der eben beschriebenen Funktionsaufrufe:

```
1  Funktion : partialSolution()

   /* Berechne den optimalen Teillösunggraphen G*^{A/O}. Diesen erhält man durch
      Einschränkung auf markierte Kanten des ODER-Spielers im expliziten
      Graphen G'^{A/O} */
2  G*^{A/O} := ⟨∅, {s_0}, ∅⟩;
3  while G*^{A/O} wächst; /* initial wahr */
4  do
       /* UND-Knoten einfügen */
5      V_NEW := { n' ∈ V'_AND | es ex. n ∈ V*_OR so dass (n,n') ∈ E' und
                                MARKED((n,n')) = true };
6      V*_AND := V*_AND ∪ V_NEW;
7      E* := E* ∪ { (n,n') ∈ E' | n ∈ V*_OR und n' ∈ V*_AND };

       /* ODER-Knoten einfügen */
8      V_NEW := { n' ∈ V'_OR | es ex. n ∈ V*_AND so dass (n,n') ∈ E' };
9      V*_OR := V*_OR ∪ V_NEW;
10     E* := E* ∪ { (n,n') ∈ E' | n ∈ V*_AND und n' ∈ V*_OR };
11 end
```

Funktion `partialSolution`

```
1  Funktion : expand(node n)

   /* Expandiere den Knoten n durch Anwendung aller zur Verfügung stehenden
      Züge. Alle dadurch erzeugten Nachfolgeknoten werden in G'^{A/O}
      aufgenommen. Sofern ein Knoten neu ist, wird die Heuristik errechnet.
      Terminale Knoten werden als gelöst markiert. */
2  if n ∈ V*_AND then
3      V_NEW := { n' ∈ V_OR | (n,n') ∈ E }\V'_OR;
4      V'_OR := V'_OR ∪ V_NEW;
5      E' := E' ∪ { (n,n') ∈ E | n' ∈ V'_OR };
6  else
7      V_NEW := { n' ∈ V_AND | (n,n') ∈ E }\V'_AND;
8      V'_AND := V'_AND ∪ V_NEW;
9      E' := E' ∪ { (n,n') ∈ E | n' ∈ V'_AND };
10 end
11 foreach n' ∈ V_NEW do
12     depth(n') := depth(n) + 1;
13     c(n') := h(n');
14     if n' ∈ R then SOLVED(n') := true else SOLVED(n') := false;
15 end
```

Funktion `expand`

```
1  Funktion : updateU()

   /* Entferne und selektiere aus U möglichst geschickt einen Knoten m. */
2  Sei m = argmax_{n∈U} depth(n);
3  U := U\{m};
4  return m;
```

Funktion `updateU`

Funktion : updateDistanceAndSolved(node m)

`/* Markiere` m `als gelöst, falls` m `ODER-Knoten und mindestens ein Kind gelöst, oder falls` m `UND-Knoten und alle Kinder gelöst. Aktualisiere die Kostenschätzung` $c(m)$`. Ist` m `ein ODER-Knoten, markiere Kante, die zum Kind mit minimaler Heuristik führt. */`

Sei $\{m_1, \ldots, m_k\} := \{\ m' \in V'_{AND} \cup V'_{OR} \mid (m, m') \in E'\ \}$;
if $m \in V'_{AND}$ **then**
  $c(m) = k + \sum_{i=1}^{k} c(m_i)$;
  $SOLVED(m) := true$;
  for $i = 1$ **to** k **do**
    if $SOLVED(m_i) = false$ **then** $SOLVED(m) := false$;
  end
else
  Sei $min := \operatorname{argmin}_{i=1,\ldots,k} c(m_i)$;
  $c(m) = 1 + c(m_{min})$;
  for $i = 1$ **to** k **do**
    $MARKED((m, m_i)) = false$;
  end
  $MARKED((m, m_{min})) = true$;
  for $i = 1$ **to** k **do**
    if $SOLVED(m_i) = true$ **then** $SOLVED(m) := true$;
  end
end

Funktion `updateDistanceAndSolved`

Funktion : selectParents(m)

`/* Wurde` m `gelöst, so überprüfe alle Väter von` m `aus` $\mathcal{G}'^{A/O}$ `in` U`. Wurde die Kostenschätzung von` m `geändert, so überprüfe alle Väter von` m `aus` $\mathcal{G}'^{A/O}$ `in` U`, die noch nicht aktualisiert wurden. */`

if $SOLVED(m)$ *wurde geändert* **then**
  $U_{NEW} := \{\ n \in V'_{AND} \cup V'_{OR} \mid (n, m) \in E'\ \}$;
else if $c(m)$ *wurde geändert* **then**
  $U_{NEW} := \{\ n \in V'_{AND} \cup V'_{OR} \mid (n, m) \in E'$ und $n \notin C\ \}$;
else
  $U_{NEW} := \emptyset$;
end
$U := U \cup U_{NEW}$;
$C := C \cup U_{NEW}$;

Funktion `selectParents`

Betrachten wir nun die Implementierung der beiden Hauptschritte von AO* etwas genauer. Hierbei beschränken wir uns auf den Code des Algorithmus AO* (Algorithmus 1), darin verwendete Funktionsaufrufe werden nur auf semantischer Ebene erläutert.

Bevor die erste Schleife (Zeile 6) betreten wird, werden zunächst einige initiale Definitionen vorgenommen. So besteht der explizite Graph initial nur aus dem ODER-Knoten s_0 (Zeile 2). Die Heuristik und Tiefe dieses Knotens werden in Zeile 3 und 4 bestimmt. Ist der Startknoten bereits ein Zielknoten, wird er als gelöst markiert (Zeile 5).

Innerhalb der äußeren Schleife (Zeile 6) wird zunächst der Vorwärtsschritt umgesetzt, der dafür verantwortlich ist, den vielversprechendsten Knoten zu expandieren. Hierzu wird vom aktuellen expliziten Graphen der optimale Teillösungsgraph bestimmt (Zeile 7), um aus diesem einen noch nicht expandierten und noch nicht gelösten Knoten mit bester Heuristik zu selektieren (Zeile 8 und 9). Den optimalen Teillösungsgraphen erhält man, indem man rekursiv in jedem Knoten entscheidet, welche Kanten (und damit auch, welche Kindknoten) des expliziten Graphen man in den optimalen Teillösungsgraphen übernimmt. Ist der aktuelle Knoten ein UND-Knoten, werden alle Kanten übernommen, ist er hingegen ein ODER-Knoten, werden nur diejenigen Kanten übernommen, die zuvor markiert wurden. Es wird immer diejenige ODER-Kante markiert, deren Kind die kleinste Kostenschätzung aufweist. Schließlich wird der eben gewählte und damit vielversprechendste Knoten expandiert, indem seine Kanten und Kinder in den expliziten Graphen installiert werden (Zeile 10).

Nun betrachten wir den Rückwärtsschritt, in welchem die Kostenschätzungen und Lösungsstatus aktualisiert werden.
Hierzu ist es zunächst erforderlich, eine Updatemenge U anzulegen, in welche all jene Knoten aufgenommen werden, die auf Aktualisierung zu überprüfen sind. Diese Menge, zusammen mit einer Kontrollmenge C, wird in Zeile 11 angelegt. In diese Kontrollmenge werden alle Knoten eingefügt, die auch in U eingefügt werden, um unendliches Aktualisieren zu verhindern. Der Rückwärtsschritt ist abgeschlossen, sobald für jeden zu überprüfenden Knoten diese Prüfung abgeschlossen ist. Dies ist der Fall, sobald die Updatemenge U leer ist, was in Zeile 12 überprüft wird. Solange diese Menge nicht leer ist, muss nach und nach ein weiterer Knoten aus dieser Menge überprüft und gegebenenfalls aktualisiert werden. Die Reihenfolge, in der Knoten aus dieser Menge ausgewählt werden, ist maßgeblich für die Lösungsqualität und die Terminierungsgeschwindigkeit verantwortlich. In Zeile 13 wird also ein Knoten aus U selektiert. Auf die verwendete Selektionsstrategie wird nachfolgend mit einem Beispiel eingegangen. Der selektierte Knoten wird in Zeile 14 auf Aktualisierung überprüft. Zunächst wird getestet, ob er als gelöst markiert werden darf. Dies ist der Fall, falls der zu untersuchende Knoten ein UND-Knoten ist und alle Kind-Knoten gelöst sind, oder falls er ein ODER-Knoten ist und mindestens ein Kind-Knoten gelöst ist. Zusätzlich muss gegebenenfalls der Wert der Kostenschätzung aktualisiert werden. Die Kostenschätzung errechnet sich im Falle des ODER-Knotens über 1 plus die minimale Kostenschätzung der Kinder und im Falle eines UND-Knotens über die Anzahl der Kinder plus die Summe ihrer Einzelschätzungen. Träten keine Zykel und Transpositionen auf, entspräche diese Kostendefinition der Anzahl der Kanten des (Teil-)Lösungsgraphen. Wurde ein Knoten tatsächlich aktualisiert, werden dessen Väter in U übernommen, abhängig davon, ob der Knoten bereits aktualisiert wurde und abhängig davon, ob die Kostenschätzung oder der Lösungsstatus aktualisiert wurde. Diesem Zusammenspiel der Bedingungen und der verwendeten Selektionsstrategie widmen wir uns nun genauer anhand eines Beispiels.

1.3.2 Beispiel zum AO*-Algorithmus

Der Graph $\mathcal{G}^{A/O}$ und damit auch $\mathcal{G}'^{A/O}$ ist im Allgemeinen nicht zyklenfrei. Dies hat zur Folge, dass der Aktualisierungsprozess der Kostenschätzungen unter Umständen nicht terminiert, falls tatsächlich Zyklen vorlieren und immer alle Väter eines aktualisierten Knotens in die Updatemenge U übernommen werden. Es gibt den Ansatz, dieses Problem durch Value- bzw. Policy-Iteration zu lösen [8]. Wir verfolgen in dieser Arbeit eine einfachere Lösungsstrategie, die darauf beruht, jeden Knoten höchstens einmal aktualisieren zu wollen. Es ist offensichtlich, dass dadurch Terminierung gewährleistet wird. Betrachten wir für diese Idee den linken Graphen aus Abbildung 1. Abbildung 2 zeigt, auf welche Weise dieser Graph durch eine Suche entstanden sein kann.
Die zuletzt durchgeführte Expansion ist die des Knotens c. Durch dessen Expansion wurde Knoten e mit Heuristik 8 erzeugt und installiert. Die Kostenschätzung des Knotens c kann damit auf 9 herabgesetzt werden. Aufgrund seiner Aktualisierung müssen Knoten a und d in U übernommen werden. Wir erinnern uns, dass wir die Strategie verfolgen, jeden Knoten nur einmal in U aufzunehmen, um unendliches Aktualisieren zu verhindern. Wir testen daher, ob die Knoten a und d in der Kontrollmenge C enthalten sind, die alle Knoten enthält, die bereits in U aufgenommen wurden. Beide Knoten waren noch nicht in C, damit auch nicht in U, womit beide in U und C aufgenommen werden können.
Nun stellt sich die Frage der Selektionsstrategie, da U mehr als einen Knoten enthält. Wir rechnen dieses Beispiel für zwei Strategien durch, um die schlussendlich verwendete zu motivieren.
Die Selektionsreihenfolge (a, d, b) führt dazu, dass die Kostenschätzung von a zu Beginn auf 23 gesenkt wird. Nach Aktualisierung von d und b könnte $c(a)$ noch weiter verbessert werden durch Herabsetzen der Kostenschätzung auf 22. Da a bereits in U aufgenommen wurde und sich damit in der Menge C befindet, darf dessen Kostenschätzung nicht erneut aktualisiert werden. Wäre hingegen die Selektionsreihenfolge (d, b, a) gewählt worden, würde $c(a)$ trotz einmaliger Aktualisierung korrekt gesetzt werden. An dieser Stelle sei vermerkt, dass selbst die als korrekt bezeichnete Kostenschätzung 22 *nicht zulässig* ist, da sie überschätzt. Unter der Voraussetzung, dass der Heuristikwert 8 des Knotens e den tatsächlichen Kosten entspricht, um e zu lösen, ist der Wert $c(a) = 22$ noch immer zu hoch, da die tatsächlichen Kosten (die Anzahl der Kanten) des optimalen Lösungsgraphen 13 betragen. Zugunsten schnellerer Laufzeit wird diese Überschätzung in Kauf genommen.
Die zweite Selektionsreihenfolge kann man durch Selektion des Knotens maximaler Tiefe erhalten. Die in dieser Arbeit gewählte Selektionsstrategie besteht also aus dem Selektieren eines Knotens maximaler Tiefe. Da der Graph nicht zyklenfrei ist, ergibt sich erneut die Problematik, wie die Tiefe zu definieren ist. Wieder wird Suboptimalität in Kauf genommen, indem jedem Knoten die Tiefe zugewiesen wird, die als erstes errechnet werden kann. Sobald ein neuer Knoten in den expliziten Graphen übernommen wird, errechnet sich seine Tiefe als 1 plus die Tiefe des Vaterknotens, durch dessen Expansion der neue Knoten erstellt wurde. Diese Tiefe wird nicht mehr aktualisiert, um Terminierung zu gewährleisten.
Die verwendete Selektionsstrategie, zusammen mit der suboptimalen Berechnung der Tiefe kann dazu führen, dass in Einzelfällen nicht in der gewünschten, optimalen Reihenfolge selektiert wird. Betrachten wir nun den rechten Graphen aus Abbildung 1 und nehmen an, dass trotz der Selektionsstrategie, die stets den Knoten maximaler Tiefe selektiert, die Selektionsreihenfolge (a, d, b) vorliegt. Der rechte Graph entspricht dem linken, lediglich Knoten e unterschiedet sich: Dieser zeichnet sich nicht mehr dadurch aus, dass seine Heuristik 8 beträgt, sondern dadurch, dass er als gelöst markiert ist. In Abbildung 1 illustrieren wir den Sachverhalt, dass Knoten e gelöst ist, durch eine doppelte Umrahmung dieses Knotens. Trotz der Heuristik $9 \neq 0$ kann dieser Knoten

gelöst sein, da er aufgrund von Transpositionen an anderer Stelle im Graphen bereits gelöst worden sein konnte. Wird a als erstes aktualisiert, bleibt sein Lösungsstatus unverändert, da er ein UND-Knoten ist und Knoten b noch nicht als gelöst markiert wurde. Nach Abarbeitung der Knoten d und b wird b aber schließlich als gelöst markiert, d.h. nun könnte auch a als gelöst markiert werden. Erlauben wir nun nicht, a erneut in U aufzunehmen, ist die Korrektheit des Algorithmus verletzt.
Damit erklärt sich die Fallunterscheidung der Funktion *selectParents*(), die, sofern ein Knoten auf gelöst gesetzt wurde, auch dann alle Eltern des Knotens in U aufnimmt, falls diese bereits aktualisiert wurden. Wurde hingegen lediglich eine Kostenschätzung aktualisiert, dürfen, wie bereits besprochen, nur Knoten in U übernommen werden, die noch nicht aktualisiert wurden. Diese Vorgehensweise garantiert Terminierung, da jeder Knoten maximal einmal von ungelöst ($SOLVED(n) = false$) auf gelöst ($SOLVED(n) = true$) springen kann, hingegen niemals von gelöst auf ungelöst.

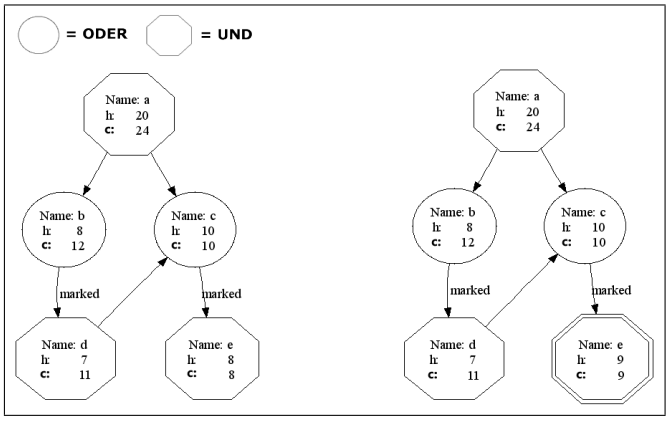

Abbildung 1: Graphen nach Expansion eines Knotens und vor der Aktualisierung.

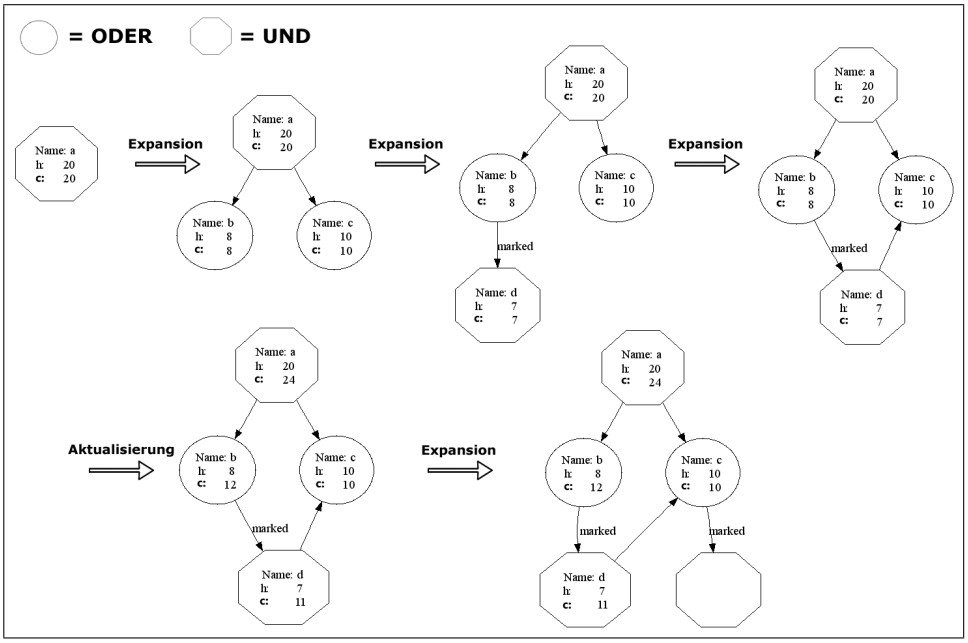

Abbildung 2: Verlauf, wie die Graphen aus Abbildung 1 entstanden sein könnten.

1.4 Die Heuristik

Da eine Heuristik die Distanz zwischen Zuständen schätzt, ist es zunächst erforderlich, den Zustandsraum genau zu definieren und insbesondere, auf welche Art und Weise ein Zug eines Spielers zwei Zustände ineinander überführt. Wir definieren eine Instanz des Erreichbarkeitsspiels auf Grundlage des STRIPS-Formalismus [6, 4].

Da wir ab jetzt ein Erreichbarkeitsspiel in Anlehnung an den STRIPS-Formalismus betrachten, verwenden wir fortan die entsprechende Notation. Züge werden daher als Aktionen bezeichnet.

Definition 11 (Instanz des Erreichbarkeitsspiels im STRIPS-Formalismus)**.**
Zunächst die Definition aller Komponenten der *Instanz der Spielstruktur im STRIPS-Formalismus.*

- Sei P eine endliche Menge von Zustandsvariablen, also eine Menge beliebiger unterscheidbarer Tokens mit $1 \notin P$ und $2 \notin P$. Jeder Zustand $s \in S$ des Erreichbarkeitsspiels $\mathcal{G}$ ist nun eine Teilmenge von P, vereinigt entweder mit der Menge $\{1\}$ oder mit der Menge $\{2\}$. Damit gilt $(s \backslash \{1,2\}) \subseteq P$. $s \backslash \{1,2\}$ entspricht dabei dem eigentlichen Zustand, während die Tokens 1 und 2 kodieren, welcher Spieler am Zug ist (siehe Definition der Abbildung p). Damit ist die Menge aller Zustände $S := \{\ s \cup \{i\} \mid s \subseteq P,\ i \in \{1,2\}\ \}$. Sei $S' := \{\ s \mid s \subseteq P\ \}$. Dann gilt $S' = 2^P$.

- Die Menge *MOVES* ist die Menge aller Aktionen, die den Spielern zur Verfügung stehen. Dabei gehört jede Aktion genau einem Spieler. Hierzu definieren wir das Komplement von i durch $\bar{i} := 3 - i$.
  Eine Aktion $m \in MOVES$ hat die Form $\langle pre, add, del \rangle$ mit $i \in \{1,2\}$ und:
  - $pre \subseteq (P \cup \{i\})$, der Precondition-List,
  - $add \subseteq (P \cup \{\bar{i}\})$, der Add-List,
  - $del \subseteq (P \cup \{i\})$, der Delete-List.

  Wir setzen eine Normalisierung voraus, d.h. es gilt $add \cap del = \emptyset$.
  Weiterhin fordern wir, dass i bzw. $\bar{i}$ in allen Komponenten tatsächlich enthalten sind, d.h. $i \in pre$, $\bar{i} \in add$ und $i \in del$.
  Der Effekt einer Aktion m kann der Transitionsfunktion δ entnommen werden.

- Die Zugzuweisungsfunktion $\Gamma_i(s)$ ist für $s \in S$ und $i \in \{1,2\}$ eine Teilmenge von *MOVES*. Es gilt $\Gamma_i(s) := \{\ m \in MOVES \mid i \in pre \subseteq s\ \}$. $\Gamma_i(s)$ weist somit jedem Spieler in jedem Zustand die Menge der ausführbaren Aktionen zu.

- Die partielle Transitionsfunktion $\delta : S \times MOVES \to S$ weist jedem Zustand $s \in S$ und jedem Zug $\gamma \in \Gamma_i(s)$ für $i \in \{1,2\}$ den Nachfolgezustand zu. Es gilt: $\delta(s, \gamma) := (s \cup add) \backslash del$, falls $\gamma = \langle pre, add, del \rangle \in \Gamma_i(s)$ und $i \in \{1,2\}$.

- Die Abbildung $p : S \to \{1,2\}$ mit $p(s) = i$ genau dann, wenn $i \in s$ für $i \in \{1,2\}$ und $s \in S$, weist jedem Zustand den Spieler zu, der am Zug ist.

Wir erinnern uns an die Forderung: Für alle $s \in S$ gilt, dass $\Gamma_i(s) = \emptyset$ für genau ein $i \in \{1,2\}$. Dies erfüllt unsere Instanz des Erreichbarkeitsspiels, da aufgrund der Konstruktion der Aktionen jeder Zustand entweder die 1 oder die 2 enthält. Da weiterhin auch jede Aktion in der Precondition-List entweder die 1 oder 2 enthält, ist durch die Definition $\Gamma_i(s) = \{\ m \in MOVES \mid i \in pre \subseteq s\ \}$ gewährleistet, dass für alle $s \in S$ gilt, dass $\Gamma_i(s) = \emptyset$ für genau ein $i \in \{1,2\}$.

Wir ergänzen nun die Spielstruktur um die noch fehlenden Komponenten. Eine *Instanz des Erreichbarkeitsspiels im STRIPS-Formalismus* besteht aus einer Instanz der Spielstruktur im STRIPS-Formalismus und den folgenden Komponenten:

- Einem Startzustand $s_0 \in S$. Wir fordern $1 \in s_0$, um diesen Zustand für den ODER-Spieler zu kennzeichnen.

- Einer nichtleeren Menge R an Zielzuständen mit $R \subseteq S$.
  Da es für uns keine Rolle spielt, welcher Spieler im Zielzustand am Zug ist, impliziert das Vorkommen eines Zustands $r \in R$ mit $i \in r$ für $i \in \{1, 2\}$, dass $r' \in R$ mit $\bar{i} \in r'$ und $r \backslash \{1, 2\} = r' \backslash \{1, 2\}$.

1.4.1 Einleitung

Als Heuristik benutzen wir die Heursitik des Fast-Forward-Planning-Systems [12, 13], kurz: FF-Heuristik. Das FF-Planning-System spezifiziert nicht nur die Heuristik, sondern auch den Suchalgorithmus. Da wir als Suchalgorithmus AO* verwenden, beschränken wir uns auf die Heuristik dieses Ansatzes.
Die FF-Heuristik stellt eine Erweiterung der HSP-Heuristik [2] dar. Beide Heuristiken lösen eine relaxierte Form des Problems, indem die Delete-Lists der Aktionen ignoriert werden. Besteht das Problem aus dem Finden einer Aktionsfolge, die einen Startzustand in einen Zielzustand überführt, wird dies in der relaxierten Form im Allgemeinen deutlich schneller gelöst, da die relaxierten Aktionen nur Zustandsvariablen hinzufügen aber nicht entfernen, wodurch man sich einem Zielzustand deutlich schneller nähern kann.
Sowohl die FF-Heuristik, als auch die HSP-Heuristik berechnet zunächst eine Menge von Aktionen, die das relaxierte Problem lösen. Die Anzahl dieser Aktionen entspricht dem Heuristikwert. Die FF-Heuristik unterscheidet sich von der HSP-Heuristik nun dadurch, dass sie nicht alle diese Aktionen mitzählt und daher eine schärfere Abschätzung der Kosten vornimmt. Da nur solche Aktionen aus dem gefundenen Plan weggelassen werden, für die gilt, dass die restlichen Aktionen das relaxierte Problem noch immer lösen, ist die FF-Heuristik immer maximal so groß wie die HSP-Heuristik. Dennoch ist die FF-Heuristik, genau wie die HSP-Heuristik, nicht *zulässig*, d.h. es ist nicht immer gewährleistet, dass sie die Kosten der optimalen Gewinnstrategie nicht überschätzt.

Um die Heuristik schnell berechnen zu können, wird eine Idealisierung vorgenommen, die als Hauptansatzpunkt für Verbesserungen angesehen werden kann (siehe Kapitel 2: Erweiterungen). Die FF-Heuristik trifft die Annahme, dass es nur einen Spieler gibt. Um dies zu erreichen, werden alle Aktionen des Spielers 2 dem Spieler 1 zugeschrieben. Damit wird implizit angenommen, dass beide Spieler kooperieren, da die Aktionen des Spielers 2 von Spieler 1 gespielt werden und damit auf eine Weise, die Spieler 1 zugute kommt. Da Spieler 2 aber der Gegenspieler von Spieler 1 ist, müsste Spieler 1 auf *alle* Aktionen des Spielers 2 reagieren können, statt *eine* dieser Aktionen zu verwenden, die für den Plan am passendsten ist.
Eine weitere Vereinfachung, die sich aus dem Verschmelzen der beiden Spieler ergibt, ist das Ignorieren der UND/ODER-Struktur. Sobald Spieler 2 zum Zug kommt, steigt der Verzweigungsgrad, da Spieler 1 auf alle gespielten Aktionen reagieren können muss. Damit ergäbe sich für jeden UND-Knoten eine Kostenschätzung zum Ziel, die (in etwa) der Summe der Kostenschätzungen aller Kindknoten entspricht. Die FF-Heuristik ignoriert diesen Sachverhalt, d.h. ein Heuristikwert eines Knotens wird für UND-Knoten und für ODER-Knoten auf dieselbe Art und Weise berechnet.
Trotz diesen Idealisierungsannahmen konnten vernünftige Ergebnisse erzielt werden (siehe Kapitel 4: Ergebnisse).

Da Aktionen nun keinem bestimmten Spieler mehr gehören und die Delete-Lists ignoriert werden, drücken wir Aktionen der Form $\langle pre, add, del \rangle = \langle \{i, p_1, \ldots, p_n\}, \{\bar{i}, a_1, \ldots, a_m\}, \{i, d_1, \ldots, d_k\} \rangle$ mit $i \in \{1, 2\}$ durch $(p_1, \ldots, p_n \rightarrow a_1, \ldots, a_m)$ aus. Dabei bezeichnen wir $\{p_1, \ldots, p_n\}$ als lhs (left hand side) und $\{a_1, \ldots, a_m\}$ als rhs

(right hand side). Eine solche idealisierte Aktion bezeichnen wir als Regel. Die Menge aller Regeln nennen wir Regelmenge und kürzen sie ab durch $RULES$.

1.4.2 Vorgehensweise der FF-Heuristik

Betrachten wir vor der algorithmischen Beschreibung der Heuristik zunächst ihre Vorgehensweise. Im Wesentlichen gibt es zwei Schritte:

In einem Vorwärtsschritt (Zeile 2 bis 9) wird nach einer oberen Grenze der Anzahl an Regeln gesucht, die eine Lösung des Problems darstellen. Sei $S[i]$ die Menge aller Zustandsvariablen des Zustands s_i. $A[i]$ sei die Menge aller Aktionen, die in $S[i]$ anwendbar sind. Für jede Schicht i nehmen wir in $S[i+1]$ alle Zustandsvariablen auf, die durch Anwendung aller Aktionen in $A[i]$ gewonnen werden. $A[i+1]$ wird analog zu $A[i]$ gewonnen. Durch wiederholte Anwendung dieser Vorgehensweise wird (sofern eine Lösung existiert) in Schicht m, mit $m \leq \min\{|RULES|, |P|\}$, eine Menge von Zustandsvariablen $S[m]$ gewonnen, so dass $S[m]$ mindestens alle Elemente eines Zielzustandes enthält. Es gilt also $r \subseteq S[m]$ für irgendein $r \in R$.

In einem Rückwärtsschritt (Zeile 10 bis 23) wird die gefundene obere Grenze der Anzahl an Regeln (die Summe aller bislang verwendeten Regeln) weiter verfeinert, indem vom Zielzustand ausgehend Regeln ausgewählt werden, bis der Startzustand erreicht wird. Wir definieren eine neue Menge $G[i]$, die alle Zustandsvariablen enthält, die wir in Zustand i gerne gewonnen haben möchten. Damit gilt $S[m] = s_m \supseteq r = G[m]$. Initial definieren wir $G[i] = \emptyset$ für alle $i \neq m$. Beginnend mit Schicht $m-1$ wird für jedes $g \in G[m]$ untersucht, ob $g \in S[m-1]$. Ist dies der Fall, konnten wir g bereits in einem früheren Schritt gewinnen und es wird somit in $G[m-1]$ übernommen. War dies nicht der Fall (also $g \notin S[m-1]$), so wird eine vollständige Voraussetzung (Precondition-List) einer beliebigen Regel aus $A[m-1]$ in $G[m-1]$ übernommen (Zeile 20) und die verwendete Regel in die Menge $SA[m-1]$ (selected actions) übernommen, der Menge aller tatsächlich verwendeten Regeln in Schicht $m-1$. Diese Vorgehensweise wird wiederholt, bis im Startzustand alle Regeln berechnet wurden, die benötigt werden, um den Zielzustand herzuleiten.

Der angegebene Algorithmus liefert lediglich einen Plan (Zeile 24), der zum Zielzustand führt, da es trivial ist, aus diesem die Heuristik abzuleiten. Der Heuristikwert des Plans $(SA[0], \ldots, SA[m-1])$ lautet $\sum_{i=0}^{m-1} |SA[i]|$.
Um eine if-Anweisung zu sparen definieren wir in Zeile 3 $S[-1] := s_0$ und in Zeile 4 $A[-1] := \emptyset$. Da $A[-1]$ die leere Menge ist, wird in Zeile 6 $S[0]$ korrekt durch s_0 definiert.

Die Funktion **selectAction(layer** i**, statevariable** g**)** (Aufruf in Zeile 19 und 20) verdient eine genauere Betrachtung. Wir widmen uns ihr in Beispiel 1.4.4.

1.4.3 Die FF-Heuristik

Algorithmus : FF-Heuristik

input : Menge von Regeln $RULES$ der Form $(lhs \rightarrow rhs)$
Startzustand s_0,
nichtleere Menge von Zielzuständen R
output : Eine Folge von Regeln, die ein $r \in R$ erzeugen

```
/* Vorwärtsschritt */
i = 0;
S[−1] := s_0;
A[−1] := ∅;
while r ⊈ S[i] für alle r ∈ R do
    /* Berechnung aller Fakten in Schicht i */
    S[i] := S[i − 1] ∪ { p ∈ rhs | (lhs → rhs) ∈ A[i − 1] };
    /* Berechnung aller Aktionen, die auf S[i] anwendbar sind */
    A[i] := { (lhs → rhs) ∈ RULES | lhs ⊆ S[i] };
    /* Gehe zur nächsten Schicht über */
    i++;
end
/* Rückwärtsschritt */
m := i − 1;
G[m] := r;
for j = m − 1 to 0 do
    G[j] := ∅;
    SA[j] := ∅;
    /* Selektiere für jede Zustandsvariable, die wir gerne in der nächsten
       Schicht hätten, eine Regel, um diese herzuleiten, falls notwendig. */
    foreach g ∈ G[j + 1] do
        if g ∈ S[j] then
            G[j] := G[j] ∪ {g};
        else
            SA[j] := SA[j] ∪ { selectAction(j, g) };
            G[j] := G[j] ∪ { p ∈ lhs | (lhs → rhs) = selectAction(j, g) };
        end
    end
end
/* Gebe Plan zurück */
return (SA[0], . . . , SA[m − 1]);
```

Algorithmus 2 : FF-Heuristik

Funktion : selectAction(layer i, statevariable g)

```
/* Selektiere geschickt eine Aktion aus A[i], die g erzeugt */
return (lhs → rhs) ∈ A[i], so dass g ∈ rhs
```

Funktion `selectAction`

1.4.4 Beispiel zur FF-Heuristik

Bei genauer Betrachtung der Berechnungsvorschrift der $A[i]$ und $S[i]$ erkennt man, dass $A[0] \subseteq \cdots \subseteq A[m]$ und $S[0] \subseteq \cdots \subseteq S[m]$. Zur besseren Lesbarkeit schreiben wir daher in diesem Beispiel $S[i] \backslash S[i-1]$, statt $S[i]$. Für $A[i]$ analog.

Es sei $RULES := \{ (A \to B) , (A \to C) , (A \to D) , (B, C \to Goal) , (D \to Goal) \}$, $s_0 := \{A\}$ und $R := \{\{Goal\}\}$.

Vorwärtsschritt:

$$
\begin{aligned}
S[0] &= \{ A \} \\
A[0] &= \{ (A \to B) , (A \to C) , (A \to D) \} \\
S[1] &= \{ B , C , D \} \\
A[1] &= \{ (B, C \to Goal) , (D \to Goal) \} \\
S[2] &= \{ Goal \} \\
A[2] &= \emptyset \text{ (Menge wird berechnet aber nicht verwendet)}
\end{aligned}
$$

Rückwärtsschritt:

$$
\begin{aligned}
G[2] &= \{ Goal \} \text{ (wird nicht berechnet, sondern definiert durch } r \in R) && (1) \\
SA[1] &= \{ (B, C \to Goal) \} && (2) \\
G[1] &= \{ B , C \} && (3) \\
SA[0] &= \{ (A \to B) , (A \to C) \} && (4) \\
G[0] &= \{ A \} && (5)
\end{aligned}
$$

Extrahierter Plan: $(\{ (A \to B) , (A \to C) \}, \{ (B, C \to Goal) \})$
Heuristikwert: 3

Im Vorwärtsschritt hat der Algorithmus keine Freiheitsgrade: Bei gegebener Menge von Regeln, einem Anfangszustand und einer Menge von Zielzuständen sind die berechneten Mengen $S[i]$ und $A[i]$ immer eindeutig.
Im Rückwärtsschritt hängt die Wahl der selektierten Regeln von der Realisierung der Funktion **selectAction(layer i, statevariable g)** ab. So wurde z.B. in Berechnungsschritt 2 und 3 die Regel $(B, C \to Goal)$ selektiert, obwohl die Wahl auch auf die Regel $(D \to Goal)$ hätte fallen können. Wäre die andere Regel selektiert worden, hätte sogar ein kürzerer Plan und damit eine genauere Heuristik berechnet werden können. Stets die korrekte Regel zu wählen (diejenige, die zum optimalen, also kürzesten Plan führt) ist ein schwieriges Problem, was im nächsten Abschnitt nochmals festgehalten wird.

1.4.5 Zulässigkeit der FF-Heuristik

Definition 12 (Zulässigkeit)**.**
Eine Heuristik ist *zulässig*, wenn sie die Anzahl an Regeln, um von einem Zustand n zu einem Zielzustand r zu gelangen, niemals überschätzt. Sei $h^*(n)$ die geringste Anzahl an Regeln, um den Zielzustand r zu erreichen und $h(n)$ die Anzahl an Regeln, die die Heuristik h errechnet, um den Zielzustand r zu erreichen. Dann heißt eine Heuristik h zulässig, falls $h(n) \leq h^*(n)$ für alle $n \in S$.

Da nach [3] bereits die relaxierte Version des Planungsproblems, in dem sämtliche Delete-Lists ignoriert werden, NP-schwer ist, wird in der verwendeten Heuristik (beide Versionen der FF-Heuristik, vergleiche Kapitel 2.1: Erweiterung der FF-Heuristik)

keine zusätzliche Zeit darauf verwendet, den optimalen Plan zu finden. Die Funktion **selectAction(layer** i**, statevariable** g**)**, die die Länge des Planes beeinflusst, selektiert daher eine beliebige Regel, ohne Optimalität zu gewährleisten. Es können verschiedene Selektionsstrategien gewählt werden. Die besten Ergebnisse liefert dabei die Selektionsstrategie, die stets eine Regel selektiert, die die kleinste Precondition-List besitzt. Mehr dazu in Kapitel 4: Ergebnisse. Dementsprechend ist es möglich, dass die FF-Heuristik die optimalen Kosten überschätzt, womit sie nicht zulässig ist. Damit wird nicht gewährleistet, dass der AO*-Algorithmus einen optimalen Lösungsgraphen findet. Die Optimalität leidet weiterhin darunter, dass die Aktualisierung der Kostenschätzungen c durch Transpositionen verfälscht wird, da diese hierdurch deutlich heraufgesetzt werden (vgl. Beispiel 1.3.2).

2 Erweiterungen

2.1 Erweiterung der FF-Heuristik

Die in diesem Ansatz verwendete FF-Heuristik wurde ursprünglich für klassische Planungsaufgaben entwickelt, d.h. für die effiziente Suche nach einem oder mehreren Zielzuständen in einem Suchraum. Das Ziel besteht somit darin, eine Folge von Regeln zu finden, die einen bestimmten Zustand erzeugt. Gewünschte Anforderungen sind Optimalität des Plans (kürzeste Folge von Regeln), sowie höchste Berechnungsgeschwindigkeit bzw. geringste Berechnungskomplexität.
In einem Erreichbarkeitsspiel treten hingegen zwei Spieler gegeneinander an, die im Allgemeinen dahingehend voneinander verschieden sind, dass ihnen unterschiedliche Regeln zur Verfügung stehen. Dieser Sachverhalt ging in die bisher vorgestellte Variante der FF-Heuristik lediglich durch die Verschmelzung der Regelmengen beider Spieler mit ein. Es wurde daher noch nicht ausreichend von der Trennung der beiden Regelmengen Gebrauch gemacht. Weiterhin spielte aufgrund des Verschmelzens der beiden Regelmengen nicht der Sachverhalt mit ein, dass beide Spieler abwechselnd ziehen müssen.

In diesem Kapitel wird eine Erweiterung der FF-Heuristik vorgestellt, die von der Unterscheidung der beiden Spieler Gebrauch macht und daher akkuratere Heuristikwerte für Erreichbarkeitsspiele liefert als die bisher vorgestellte Variante.

2.1.1 Vorgehensweise der erweiterten FF-Heuristik

Betrachten wir vor der algorithmischen Beschreibung der Heuristik zunächst ihre Vorgehensweise. Im Wesentlichen gibt es drei Schritte. Betrachten wir aber zunächst die Eingaben der erweiterten FF-Heuristik, da sich diese bereits von denen des ursprünglichen Algorithmus unterscheiden.

Die erweiterte Variante der FF-Heuristik besitzt als Eingabe statt lediglich einer Menge $RULES$ von Regeln, zwei Mengen $RULES_i$ für $i \in \{1, 2\}$, wobei die Menge $RULES_i$ diejenige Menge von Regeln ist, die Spieler i zur Verfügung steht. Damit gilt $RULES = RULES_1 \cup RULES_2$, wobei die Mengen $RULES_i$ nicht notwendigerweise disjunkt sein müssen. Um das abwechselnde Ziehen der beiden Spieler realisieren zu können, muss der Algorithmus neben dem Startzustand s_0 auch wissen, welcher Spieler i in diesem Zustand am Zug ist (im Algorithmus wird hierfür die Variable k verwendet). Eine weitere Eingabe ist eine nicht-leere Menge R, die alle Zielzustände des Erreichbarkeitsspiels $\mathcal{G}$ enthält.

In einem Vorwärtsschritt wird nach einer oberen Grenze der Anzahl an Regeln gesucht, die eine Lösung des Problems darstellen. Die Berechnungsvorschrift ist hierbei analog zur nicht-erweiterten Variante. Statt von Schicht zu Schicht immer alle anwendbaren Regeln aus $RULES$ zu selektieren, werden abwechselnd alle anwendbaren Regeln aus $RULES_i$ und $RULES_{\bar{i}}$ (mit $\bar{i} := 3 - i$) selektiert. Dies wird in Zeile 7 gewährleistet, da der Ausdruck $(k - 1 + i)\%2 + 1$ für die Schleifenzählvariable i und den Startspieler k immer zu derjenigen Zahl $l \in \{1, 2\}$ ausgewertet wird, für die gilt, dass Spieler l in Schicht i am Zug ist.

Der Rückwärtsschritt entspricht ebenfalls fast gänzlich dem Rückwärtsschritt der nicht-erweiterten FF-Heuristik. D.h. die gefundene obere Grenze der Anzahl an Regeln (die Summe aller bislang verwendeten Regeln) wird weiter verfeinert, indem vom Zielzustand ausgehend Regeln ausgewählt werden, bis der Startzustand erreicht wird. Der Unterschied liegt darin, dass zusätzlich in Zeile 25 die von den beiden Spielern selektierten Regeln in zwei getrennte Mengen SA_1 und SA_2 abgelegt werden. Zwar sind diese Regeln bereits in dem Mengen-Array $SA[]$ enthalten, allerdings verteilt über jeden zweiten Array-Eintrag. Das zusätzliche Ablegen in getrennte Mengen erleichtert spätere Abfragen.

Der dritte Schritt ist nun dafür verantwortlich, aus dem gefundenen Plan die Heuristik zu extrahieren. Hierzu finden im Wesentlichen zwei Ideen Anwendung:

1. Nicht jeder gefundene Plan spiegelt das abwechselnde Ziehen von Regeln korrekt wieder. Enthält der gefundene Plan beispielsweise n Regeln des ersten Spielers, aber keine Regeln des zweiten und nehmen wir weiter an, dass nicht nur keine Regeln des zweiten Spielers selektiert wurden, sondern dass dieser tatsächlich keine Regeln zum Plan beitragen *kann*, so ist der korrekte Heuristikwert $2n-1$, da aufgrund des abwechselnden Ziehens der zweite Spieler noch $n-1$ Regeln ausführen muss, selbst wenn diese nicht zum gefundenen Plan beitragen. Dieser Schritt ist folglich dafür verantwortlich, die gefundene Anzahl an Regeln zu erhöhen, um einen akkurateren Heuristikwert zu bestimmen. Das Erhöhen der Heuristik um einen Wert kann abstrakt als das Auffüllen des Plans mit no-Operations angesehen werden, wann immer der Spieler, der weniger Regeln zum Plan beiträgt am Zug ist und bereits alle zum Plan beitragenden Regeln gespielt hat.

2. Die zweite Erweiterung ist mehr technischer Natur: Der Algorithmus selektiert in jeder Schicht immer alle Regeln, die von dem Spieler, der gerade am Zug ist, gespielt werden können. Dabei ist nicht auszuschließen, dass auch der andere Spieler hätte Regeln spielen können, die später im gefundenen Plan Verwendung finden. Selektiert der Algorithmus (zum Beispiel bereits im ersten Schritt) n Regeln für Spieler i und löst damit das Spiel, liefert er unter Einbeziehung der ersten vorgestellten Erweiterung, in jedem Fall den Heuristikwert $2n-1$ zurück. Diese Schätzung ist aber zu hoch, falls Spieler 2 auch hätte Regeln zum Plan beitragen können. Die zweite Erweiterung ist damit dafür verantwortlich, herauszufinden, ob selektierte Regeln des Spielers i auch von Spieler $\overline{i}$ hätten gespielt werden können, um auf diese Weise zu versuchen, beide Spieler möglichst gleich viele Regeln zum Plan beitragen zu lassen.

Betrachten wir nun die algorithmische Umsetzung dieser beiden Ideen:
In Zeile 27 wird in der Variablen *max* der Spieler abgelegt, der die echte Mehrheit an Regeln zum Plan beiträgt. Analog wird in Zeile 28 der andere Spieler in der Variablen *min* abgelegt. Diese beiden Spieler werden von nun an als Maxspieler und Minspieler bezeichnet. In den Zeilen 30 und 29 werden die Kardinalitäten der selektierten Regelmengen gespeichert, die in Zeile 25 erzeugt wurden.
Einen idealen Heuristikwert erhält man, falls beide Spieler gleich viele Regeln zum Plan beitragen, da in diesem Fall keine no-Operations in den Plan eingefügt werden müssen. Wir möchten daher versuchen, so lange Regeln aus der selektierten Regelmenge SA_{max} des Maxspielers in die selektierte Regelmenge des Minspielers zu verschieben, bis beide Mengen gleich groß sind. Zu diesem Zweck wird in Zeile 31 über jede Regel aus SA_{max} iteriert. In Zeile 33 bis 36 wird getestet, ob diese Regel vom Minspieler hätte gespielt werden können. Falls dies zutrifft, wird der Zähler, der für die Anzahl der selektierten Regeln des Maxspielers steht, dekrementiert und der Zähler des Minspielers entsprechend inkrementiert. Zeile 32 stellt die Abbruchbedingung dar. Es darf nicht nur dann abgebrochen werden, wenn beide selektierte Regelmengen gleich groß

werden, sondern auch dann, wenn der Startspieler eine Regel mehr zum Plan beiträgt als der Nichtstartspieler, da auch in diesem Falle keine no-Operations eingefügt werden müssen. Um diesen Spezialfall abzufangen, wird zum Zähler des Minspielers der Wert $max\%2$ addiert, da dieser Wert genau dann zu 1 auswertet, falls der Maxspieler gleichzeitig dem Startspieler entspricht.
Nachdem über alle selektierten Regeln des Maxspielers iteriert wurde, ist nun gewährleistet, dass der Plan optimal ausgewogen ist, d.h. dass beide Spieler möglichst gleich viele Regeln zum Plan beitragen. In Zeile 38 bis 42 wird nun der Heuristikwert berechnet. Dieser entspricht der maximalen Anzahl an Regeln, die ein Spieler zum Plan beiträgt, multipliziert mit Faktor 2, um das abwechselnde Ziehen zu repräsentieren. Trägt der Startspieler mehr Regeln zum Plan bei als der Nichtstartspieler, so kann von diesem Wert 1 abgezogen werden, was an einem geeigneten Beispiel leicht ersichtlich wird.

2.1.2 Die erweiterte FF-Heuristik

Algorithmus : erweiterte FF-Heuristik

input : Menge von Regeln $RULES_1$ der Form $(lhs \rightarrow rhs)$,
Menge von Regeln $RULES_2$ der Form $(lhs \rightarrow rhs)$,
Startzustand s_0,
Startspieler $k \in \{1, 2\}$ mit $\bar{k} := 3 - k$,
nichtleere Menge von Zielzuständen R
output : Eine Folge von Regeln, die ein $r \in R$ erzeugen

```
/* Vorwärtsschritt */
```
2 $i = 0$;
3 $S[-1] := s_0$;
4 $A[-1] := \emptyset$;
5 **while** $r \notin S[i]$ *für alle* $r \in R$ **do**
  `/* Berechnung aller Fakten in Schicht` i `*/`
6 $S[i] := S[i-1] \cup \{\ p \in rhs \mid (lhs \rightarrow rhs) \in A[i-1]\ \}$;
  `/* Berechnung aller Aktionen, die auf` $S[i]$ `anwendbar sind */`
7 $A[i] := \{\ (lhs \rightarrow rhs) \in RULES_{(k-1+i)\%2+1} \mid lhs \subseteq S[i]\ \}$;
  `/* Gehe zur nächsten Schicht über */`
8 $i{+}{+}$;
9 **end**

Algorithmus 3 : erweiterte FF-Heuristik **(Teil 1)**

```
   /* Rückwärtsschritt */
10 SA_k = ∅;                    /* selektierte Regelmenge des Startspielers k */
11 SA_k̄ = ∅;                   /* selektierte Regelmenge des zweiten Spielers k̄ */
12 m := i − 1;
13 G[m] := r;
14 for j = m − 1 to 0 do
15     G[j] := ∅;
16     SA[j] := ∅;
       /* Selektiere für jede Zustandsvariable, die wir gerne in der nächsten
          Schicht hätten, eine Regel, um diese herzuleiten, falls notwendig. */
17     foreach g ∈ G[j + 1] do
18         if g ∈ S[j] then
19             G[j] := G[j] ∪ {g};
20         else
21             SA[j] := SA[j] ∪ { selectAction(j, g) };
22             G[j] := G[j] ∪ { p ∈ lhs | (lhs → rhs) = selectAction(j, g) };
23         end
24     end
25     SA_{(k−1+j)%2+1} := SA_{(k−1+j)%2+1} ∪ SA[j];
26 end

   /* Teste, ob selektierte Regeln von anderem Spieler hätten gespielt werden
      können. Schichte um, bis beide Spieler gleich viele Regeln zum Plan
      beitragen. */
27 if |SA_k| > |SA_k̄| then max := k else max := k̄;
28 min := 3 − max;
29 counter_max := |SA_max|;
30 counter_min := |SA_min|;
31 foreach r ∈ SA_max do
32     if counter_max ≤ counter_min + max%2 then break;
33     if r ∈ RULES_min then
34         counter_min + +;
35         counter_max − −;
36     end
37 end
38 if counter_k > counter_k̄ then
39     return 2 ∗ counter_k − 1;
40 else
41     return 2 ∗ counter_k̄;
42 end
```

Algorithmus 3 : erweiterte FF-Heuristik **(Teil 2)**

2.1.3 Beispiel zur erweiterten FF-Heuristik

Regelmenge des Spielers 1: $RULES_1 := \{ (1 \rightarrow 2) , \ldots , (1 \rightarrow 8) , (9 \rightarrow 10) \}$,
Regelmenge des Spielers 2: $RULES_2 := \{ (1 \rightarrow 2) , (1 \rightarrow 3) , (8 \rightarrow 9) \}$,
Startzustand $s_0 := \{1\}$,
Menge der Zielzustände $R = \{\{1, \ldots, 10\}\}$,
Startspieler $k = 1$

Vorwärtsschritt:

$S[0] = \{ 1 \}$
$A[0] = \{ (1 \rightarrow 2) , \ldots , (1 \rightarrow 8) \}$
$S[1] = \{ 1 , \ldots , 8 \}$
$A[1] = \{ (1 \rightarrow 2) , (1 \rightarrow 3) , (8 \rightarrow 9) \}$
$S[2] = \{ 1 , \ldots , 9 \}$
$A[2] = \{ (1 \rightarrow 2) , \ldots , (1 \rightarrow 8) , (9 \rightarrow 10) \}$
$S[3] = \{ 1 , \ldots , 10 \}$
$A[3] = \{ (1 \rightarrow 2) , (1 \rightarrow 3) , (8 \rightarrow 9) \}$ (Menge wird berechnet aber nicht verwendet)

Rückwärtsschritt:

$G[3] = \{ 1 , \ldots , 10 \}$ (wird nicht berechnet, sondern definiert durch $r \in R$)
$SA[2] = \{ (9 \rightarrow 10) \}$
$G[2] = \{ 1 , \ldots , 9 \}$
$SA[1] = \{ (8 \rightarrow 9) \}$
$G[1] = \{ 1 , \ldots , 8 \}$
$SA[0] = \{ (1 \rightarrow 2) , \ldots , (1 \rightarrow 8) \}$

SA_1: $\{ (1 \rightarrow 2) , \ldots , (1 \rightarrow 8) , (9 \rightarrow 10) \}$
SA_2: $\{ (8 \rightarrow 9) \}$

Nun wurde berechnet, welcher Spieler welche Regeln zum Plan beiträgt. Spieler 1 trägt dabei 8 Regeln zum Plan bei, Spieler 2 hingegen lediglich eine. Da Spieler 1 damit beginnt, Regeln auszuführen, und beide Spieler abwechelnd ziehen, könnte man nun den Heuristikwert 15 errechnen. Dieser Wert wird im dritten Schritt angepasst, indem getestet wird, ob Spieler 2 ebenfalls hätte Regeln zum Plan beitragen können, die bislang von Spieler 1 gespielt wurden.

$max = 1$, $min = 2$.
$counter_{max} = 8$
$counter_{min} = 1$

Nach Ausführung der for-Schleife (Zeile 31 bis 37) ergeben sich folgende Werte:

$counter_{max} = 6$
$counter_{min} = 3$

Damit ergibt sich ein Heuristikwert von 11.

2.1.4 Vergleich: normale FF-Heuristik - erweiterte FF-Heuristik

Wir stellen fest, dass die erweiterte FF-Heuristik für Erreichbarkeitsspiele präzisere Heuristikwerte liefert als die normale FF-Heuristik. Gleichzeitig kann die normale FF-Heuristik niemals bessere Ergebnisse liefern als die erweiterte. Es bleibt jedoch die Frage zu klären, wie groß der Effizienzgewinn durch die Verwendung der erweiterten Variante, statt der normalen ist. Hierfür betrachten wir folgende Szenarien:

1. **Regelmenge von Spieler 1 = Regelmenge von Spieler 2**
   In diesem Szenario unterschieden sich die Heuristikwerte nicht voneinander.

2. **Regelmenge von Spieler 1 $\cap$ Regelmenge von Spieler 2 = $\emptyset$**
   In diesem Szenario ist bedeutend, ob der zweite Spieler Regeln in seiner Regelmenge zur Verfügung hat, die zu einem Plan beitragen, der das Spiel löst. Hat der zweite Spieler keine solche Regeln zur Verfügung, so liefert die erweiterte FF-Heuristik zwar präzisere Heuristikwerte, doch unterschieden sich diese von denen der normalen FF-Heuristik stets um den konstanten Faktor 2. Da dieser konstant ist, nimmt dies nur geringen Einfluss auf die Selektionsreihenfolge. Existieren hingegen Regeln des zweiten Spielers, die zu einem Plan beitragen, so liegt der Unterschied zwischen den beiden Heuristiken zwischen Faktor 1 und 2, abhängig von der Instanz des Erreichbarkeitsspiels. In diesem Fall verbessert die erweiterte Heuristik das Selektionsverhalten der Knoten.

3. **Regelmenge von Spieler 1 $\cap$ Regelmenge von Spieler 2 $\neq \emptyset$**
   Auch hier ist lediglich von Bedeutung, ob der zweite Spieler Regeln zur Verfügung hat, die zu einem Plan beitragen. Die Fallunterscheidung ist demzufolge identisch zu Fall (2).

Wir stellen also fest, dass die erweiterte Heuristik genau dann einen Effizienzgewinn gegenüber der normalen FF-Heuristik verspricht, wenn die Regelmengen der beiden Spieler nicht identisch sind und gleichzeitig der zweite Spieler Regeln besitzt, die zu einem Plan beitragen.

2.2 Modifikation des AO*-Algorithmus

2.2.1 Expansionsverhalten

Der vorgestellte AO*-Algorithmus expandiert unter allen noch nicht expandierten Knoten einen solchen, der sich in einem optimalen Teillösungsgraphen befindet (vgl. Zeile 8 und 9 in AO*) und dabei einen minimalen Heuristikwert h besitzt. Dabei wird demzufolge nicht zwischen UND- und ODER-Knoten unterschieden. Diese Vorgehensweise ist analog zur Beschreibung des AO*-Algorithmus in [14]. Sie ist sinnvoll, falls die verwendete Heuristik sehr zuverlässige Ergebnisse liefert, d.h. falls sie nur wenig von den tatsächlichen Kosten abweicht, die erforderlich sind, um den augenblicklichen Knoten zu beweisen. Da durch Verwendung der FF-Heuristik normalerweise eine sehr große Diskrepanz herrscht zwischen dem Wert der Heuristik h und der Kostenschätzung c, die eine Schätzung der Kosten angibt, die noch notwendig sind, um einen Knoten zu beweisen, ist es eine sinnvolle Vorgehensweise, für UND-Knoten die Heuristik nicht zu errechnen. Eine bessere Alternative bietet das sofortige Expandieren aller UND-Knoten bei ihrer Erzeugung. Diese Vorgehensweise spart enormen Berechnungsaufwand und zieht dennoch keine Nachteile nach sich.
Der Grund hierfür ist folgender: Wir gehen davon aus, dass die verwendete Heuristik deutlich unterschätzt, da diese lediglich einen Pfad zu einem Zielknoten sucht, ohne auf die UND/ODER-Struktur des Graphen einzugehen. Da sich die initiale Kostenschätzung eines UND-Knotens aus der Summe der Heuristiken seiner Kinder ergibt,

ist diese im Allgemeinen deutlich größer, als seine Heuristik. Betrachten wir den folgenden Graphen (hier: Baum): $\mathcal{G}'^{A/O} = \langle \{a_1, \ldots, a_n\}, \{o_1\}, \{(o_1, a_1), \ldots, (o_1, a_n)\}\rangle$.

Unter der Voraussetzung, dass die Heuristik der UND-Knoten stark unterschätzt, wird zunächst der UND-Knoten mit minimaler Heuristik, o.B.d.A sei dies a_1, expandiert. Aufgrund der Unterschätzung ist es wahrscheinlich, dass die verbleibenden UND-Knoten $a_2, \ldots, a_n$ Heuristiken besitzen, die in etwa gleich groß sind, wie die Heuristiken der durch Expansion von a_1 neu gewonnenen ODER-Knoten. Damit werden alle UND-Knoten etwa zur selben Zeit expandiert, wie die neu gewonnenen ODER-Knoten. Wären direkt nach der Expansion von o_1 alle UND-Knoten $a_1, \ldots, a_n$ expandiert worden, wären neben n Heuristikberechnungen zusätzlich noch n Berechungen der optimalen Teillösungsgraphen eingespart worden. Nehmen wir an, dass der Graph $\mathcal{G}'^{A/O}$ lediglich einen Teilgraphen eines expliziten Graphen darstellt, so ergibt sich eine Ersparnis von n Pfadberechnungen von der Wurzel s_0 des Graphen zu o_1, sowie die Ersparnis der Heuristikberechnungen der n UND-Knoten. Diese Ersparnisse können beträchtlich sein, da das Berechnen der Heuristiken teuer ist, falls die Zielzustände vom aktuellen Knoten weit entfernt sind, während das Berechnen des optimalen Teillösungsgraphen teuer ist, falls sich der aktuelle Knoten tief im expliziten Graphen befindet.

2.2.2 Kostenschätzung der UND-Knoten

Wir haben einen optimalen Teillösungsgraphen als einen Lösungsgraphen definiert, der die kleinste Anzahl an Kanten besitzt (vgl. Definition 9). Der Lösungsgraph, der vom AO*-Algorithmus zurückgeliefert wird, ist im Allgemeinen kein optimaler Lösungsgraph, da neben überschätzenden Heuristikwerten vor allem Transpositionen dazu beitragen, dass die Kostenschätzungen nicht die Anzahl der Kanten widerspiegeln, sondern einen Wert, der deutlich höher ist.
Neben dem Problem, dass wegen Transpositionen ein Lösungsgraph mit minimalen Kostenschätzungen nicht zwingend derjenige mit geringster Anzahl an Kanten ist, ergibt sich durch das Aufsummieren der Kostenschätzungen bei UND-Knoten das Problem möglicher Zahlenbereichsüberläufe. Bereits mit sehr kleinen Beispielen ist *int* nicht mehr ausreichend. Selbst der größte Ganzzahlenbereich *long* reicht unter Umständen (vgl. Kapitel 4: Ergebnisse) nicht mehr aus. Um dieses Problem zu umgehen, sind zwei Kostenschätungsberechnungen der UND-Knoten wählbar. Die erste Variante wurde bereits ausführlich behandelt und entspricht der Summation über die Kostenschätzungen der Kinder. Eine weitere Variante ist die Bildung des Maximums über alle Kostenschätzungen der Kinder.
Die zweite Variante bietet den Vorteil, dass Transpositionen den Wert des Maximums nicht mehr verfälschen. Weiter ist ein Wertebereichsüberlauf nun ausgeschlossen, da die maximale Pfadlänge den Bereich von *long* praktisch nicht überschreiten kann. Die Lösungsgraphen, die diese beiden Kostenschätzungsstrategien liefern, unterscheiden sich in ihrer Struktur wiefolgt:
Zunächst nehmen wir an, dass keine Transpositionen auftreten, oder diese zumindest keinen wesentlichen Einfluss auf die Selektionsreihenfolge nehmen, da sonst keine sicheren Aussagen über die Struktur des Lösungsgraphen gemacht werden können.

- Die Berechnung des Maximums gewährleistet einen Lösungsgraphen, der minimale Tiefe hat. Damit hat die induzierte Gewinnstrategie minimale Länge, so dass nach minimal vielen Aktionen des ODER-Spielers das Spiel gelöst wird.

- Die Berechnung der Summe gewährleistet eine möglichst kleine Anzahl an Knoten im Lösungsgraphen, während über seine Tiefe keine Aussage getroffen wird. Damit ist es möglich, dass der ODER-Spieler mehr Aktionen spielen muss, bis das Spiel gewonnen ist, während die Kodierung der Strategie aber kleiner ist, als im Falle der Maximierung der UND-Kostenschätzungen.

Abbildung 3 zeigt, dass die Summen-Kostenschätzung zu einem Lösungsgraphen führen kann, der weniger Knoten enthält, während die Maximum-Kostenschätzung den Lösungsgraphen erstellt, der zwar mehr Knoten enthält, dafür aber schneller zum Ziel führt.
Der Lösungsgraph, den man durch die Summen-Kostenschätzung erhält, besteht in Abbildung 3 aus dem Wurzelknoten, sowie dem rechten Teilgraphen. Dessen Tiefe beträgt 7, die Anzahl seiner Knoten beträgt 8. Der Lösungsgraph, den man durch die Maximum-Kostenschätzung erhält, besteht aus dem Wurzelknoten und dem linken Teilgraphen. Dessen Tiefe beträgt lediglich 4, während er 11 Knoten enthält.
Terminale Knoten, also Knoten aus R, sind durch doppelte Umrahmungen gekennzeichnet.

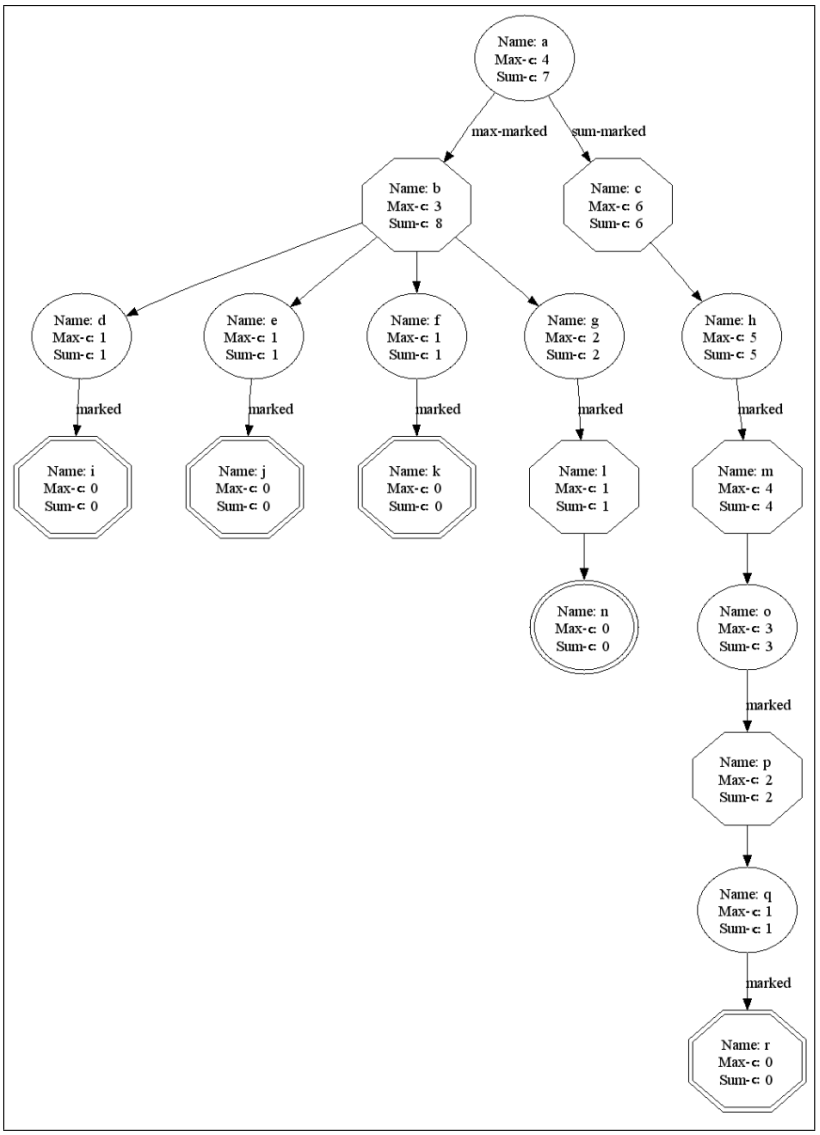

Abbildung 3: Beispiel, das den Unterschied zwischen Maximum-Kostenschätzung und Summen-Kostenschätzung zeigt.

3 Implementierung

In diesem Kapitel wird auf die Implementierung eingegangen, deren Ergebnisse im nächsten Kapitel präsentiert werden. Dabei wird zunächst das Eingabeformat vorgestellt und anschließend eine lose Ansammlung an Implementierungsdetails, die besondere Aufmerksamkeit verdienen.

3.1 Das Eingabeformat

Das Eingabeformat besteht aus zwei Textdateien, deren Schablonen in Abbildung 4 und Abbildung 6 dargestellt sind. Es wurde eine Auftrennung vorgenommen zwischen dem Transitionssystem (vgl. Definition 1: Spielstruktur), und der Aufgabe (vgl. Definition 2: Erreichbarkeitsspiel), die darauf auszuführen ist, welche die Spielstruktur zum Erreichbarkeitsspiel erweitert. Dies ermöglicht es, lediglich einmal das Transitionssystem zu formalisieren und anzugeben, um verschiedene Aufgaben (d.h. verschiedene Zielmengen R oder verschiedene Startzustände s_0 zu definieren) darauf auszuführen. Die beiden Musterdateien sind selbsterklärend. Dabei sind Zeilen, die von Sternen umschlossen sind, durch ihre Beschreibung zu ersetzen, während Zeilen, die nicht durch Sterne eingeschlossen sind, zur Syntax des Eingabeformats gehören und damit nicht verändert werden dürfen.

3.1.1 Eingabeformat der Spielstruktur

```
1  number of actions player 1:
2  *a1 (Anzahl Aktionen des Spielers 1)*
3
4  number of actions player 2:
5  *a2 (Anzahl Aktionen des Spielers 2)*
6
7  actions player 1:
8  *a1 Aktionen (in a1 Zeilen) der Form:*
9  *Aktionsname ; <pre1,...,pren ; add1,...,addm ; del1,...,delk>*
10
11 actions player 2:
12 *a2 Aktionen (in a2 Zeilen) der Form:*
13 *Aktionsname ; <pre1,...,pren ; add1,...,addm ; del1,...,delk>*
14
15 comments:
16 *Beliebig viele Kommentare in beliebig vielen Zeilen.*
17 *Zusätzlich darf hinter jeder Zeile gemäß Syntaxvorschrift nach einem*
18 *Leerzeichen und nach zwei Slashes // ein beliebiger Kommentar folgen.*
19 *Wichtig ist außerdem, dass leere Listen durch !EMPTY! kodiert werden.*
20 *Bsp.: Eine Aktion mit Voraussetzungen pre1 und pre2, die add1 hinzufügt,*
21 *aber nichts löscht, wird kodiert durch <pre1,pre2 ; add1 ; !EMPTY!>*
```

Abbildung 4: Eingabeformat der Spielstruktur

Um die Gewinnbedingung des ODER-Spielers formal definieren zu können, wurde gefordert, dass in jedem Zustand der Spieler, der gerade am Zug ist, eine Aktion spielen können muss. Diese Einschränkung gilt nicht für die vorgenommene Implementierung.

Jeder Zustand, der nicht als gewonnen markiert ist und in dem ein Spieler keine Aktion vornehmen kann, gilt für den ODER-Spieler als verloren. Diese Vorgehensweise ist sinnvoll, da es das Ziel des ODER-Spielers ist, einen beliebigen Zielzustand zu erreichen. Wurde ein Zustand erzeugt, der nicht als gewonnen markiert ist und der keine Nachfolgezustände hat, kann der ODER-Spieler das Spiel (ausgehend vom aktuellen Zustand) nicht mehr gewinnen.

3.1.1.1 Beispiel (Spielstruktur von Tic Tac Toe)

```
1  number of actions player 1:
2  9
3
4  number of actions player 2:
5  9
6
7  actions player 1:
8  P1_11 ; <f_11 ; 1_11,nf_11 ; f_11>  // place an X at position 11
9  P1_21 ; <f_12 ; 1_12,nf_12 ; f_12>
10 P1_31 ; <f_13 ; 1_13,nf_13 ; f_13>
11 P1_21 ; <f_21 ; 1_21,nf_21 ; f_21>
12 P1_22 ; <f_22 ; 1_22,nf_22 ; f_22>  // ...
13 P1_23 ; <f_23 ; 1_23,nf_23 ; f_23>
14 P1_31 ; <f_31 ; 1_31,nf_31 ; f_31>
15 P1_32 ; <f_32 ; 1_32,nf_32 ; f_32>
16 P1_33 ; <f_33 ; 1_33,nf_33 ; f_33>  // place an X at position 33
17
18 actions player 2:
19 P2_11 ; <f_11 ; 2_11,nf_11 ; f_11>  // place an O at position 11
20 P2_12 ; <f_12 ; 2_12,nf_12 ; f_12>
21 P2_13 ; <f_13 ; 2_13,nf_13 ; f_13>
22 P2_21 ; <f_21 ; 2_21,nf_21 ; f_21>
23 P2_22 ; <f_22 ; 2_22,nf_22 ; f_22>  // ...
24 P2_23 ; <f_23 ; 2_23,nf_23 ; f_23>
25 P2_31 ; <f_31 ; 2_31,nf_31 ; f_31>
26 P2_32 ; <f_32 ; 2_32,nf_32 ; f_32>
27 P2_33 ; <f_33 ; 2_33,nf_33 ; f_33>  // place an O at position 33
28
29 comments:
30 f_ab  stands for: field (a,b) is free
31 nf_ab stands for: field (a,b) is NOT free
32
33 P1_ab stands for: player 1 places his sign in field (a,b)
34 P2_ab stands for: player 2 places his sign in field (a,b)
```

Abbildung 5: Eingabeformat der Spielstruktur für Tic Tac Toe

3.1.2 Eingabeformat der Aufgabe

Das Eingabeformat der Aufgabe unterscheidet sich von der Definition des Erreichbarkeitsspiels in zwei Details. Betrachten wir aber zunächst das Eingabeformat.

```
start state:
*Ein Zustand der Form: token1,token2,...,tokenn*

number of goal states player 1:
*g1 (Anzahl der Zielzustände des Spielers 1)*

goal states player 1:
*g1 Zielzustände des Spielers 1 in g1 Zeilen*

number of goal states player 2:
*g2 (Anzahl der Zielzustände des Spielers 2)*

goal states player 2:
*g2 Zielzustände des Spielers 2 in g2 Zeilen*

comments:
*Beliebig viele Kommentare in beliebig vielen Zeilen.*
*Zusätzlich darf hinter jeder Zeile gemäß Syntaxvorschrift nach einem*
*Leerzeichen und nach zwei Slashes // ein beliebiger Kommentar folgen.*
*Wichtig ist außerdem, dass leere Listen durch !EMPTY! kodiert werden.*
*Bsp.: Eine Aktion mit Voraussetzungen pre1 und pre2, die add1 hinzufügt,*
*aber nichts löscht, wird kodiert durch <pre1,pre2 ; add1 ; !EMPTY!>*
```

Abbildung 6: Eingabeformat der Aufgabe

In der Aufgabe wurde die Menge R definiert als eine Menge von Zielzuständen, so dass jeder Zustand $s \in R$ als gewonnen gilt (vgl. Definition 2: Erreichbarkeitsspiel). Da in der Praxis auch alle Zustände s' mit $s' \supset s$ als gewonnen gelten, reicht es aus, das *minimale* s als Zielzustand anzugeben, das der Bedingung genügt, dass s' Zielzustand für alle $s' \supset s$. Einen Zustand s nennen wir dabei minimalen Zielzustand, falls $s \in R$ und $s \backslash \{p\} \notin R$ mit $p \in s$ beliebig.

Das Eingabeformat des Transitionssystems lässt neben den Zielzuständen des ODER-Spielers eine Menge von Zielzuständen für den UND-Spieler zu. Die Angabe von Zielzuständen für den UND-Spieler erleichtert ebenfalls das Formalisieren eines Systems. Eine Regel, die auf jedes System zutrifft, ist die folgende: „Jeder Spieler darf nur in solchen Zuständen von no-Operations verschiedene Aktionen ausführen, in denen noch kein Spieler gewonnen hat“. Diesen Sachverhalt müsste man bei der Formalisierung der Voraussetzungen (Precondition-Lists) der Aktionen berücksichtigen. Gibt man hingegen alle Zustände an, die für den UND-Spieler als gewonnen gelten, so kann auf diese Formalisierung verzichtet werden, da die Implementierung verhindert, dass Zustände expandiert werden, die für den UND-Spieler oder den ODER-Spieler gewonnen sind.

3.1.2.1 Beispiel (Aufgabe von Tic Tac Toe)

```
1  start state:
2  f_11,f_12,f_13,f_21,f_22,f_23,f_31,f_32,f_33
3
4  number of goal states player 1:
5  8
6
7  goal states player 1:
8  1_11,1_21,1_31  // vertical
9  1_12,1_22,1_32
10 1_13,1_23,1_33
11 1_11,1_12,1_13  // horizontal
12 1_21,1_22,1_23
13 1_31,1_32,1_33
14 1_11,1_22,1_33  // diagonal
15 1_31,1_22,1_13
16
17 number of goal states player 2:
18 8
19
20 goal states player 2:
21 2_11,2_21,2_31  // vertical
22 2_12,2_22,2_32
23 2_13,2_23,2_33
24 2_11,2_12,2_13  // horizontal
25 2_21,2_22,2_23
26 2_31,2_32,2_33
27 2_11,2_22,2_33  // diagonal
28 2_31,2_22,2_13
29
30 comments:
31 no comments!
```

Abbildung 7: Eingabeformat der Aufgabe für Tic Tac Toe.

3.2 Implementierungsdetails

3.2.1 Einschränkung der Aktionsmenge

Sobald ein Knoten n expandiert wird, wird in Zeile 5 und 9 der Funktion **expand()** jede ausgehende Kante (des impliziten Graphen) des Knotens n in den expliziten Graphen übernommen. Dabei entspricht jede Kante eines Knotens im impliziten Graphen einer im entsprechenden Zustand spielbaren Aktion.
Nun ist es möglich, dass verschiedene Aktionen in denselben Zielzustand führen. Damit wäre die Anzahl der verschiedenen Kindknoten von n kleiner als die Anzahl seiner ausgehenden Kanten. Daher ist es sinnvoll, nur solche Kanten zu übernehmen, die in unterschiedliche Kindknoten führen. Errechnet man beispielsweise die Kostenschätzung eines UND-Knotens (welche sich durch die Anzahl der ausgehenden Kanten plus die Kostenschätzungen aller Kinder ergibt), verfälscht sich dessen Schätzung beliebig, da Kindknoten mehrfach in die Berechnung eingehen.

3.2.2 Heuristikberechnung

In der Berechnung der FF-Heuristik (in beiden Versionen) dauert der Vorwärtsschritt so lange an, bis ein beliebiger Zielzustand r gefunden wurde. Für genau dieses r wird der Rückwärtsschritt durchgeführt, d.h. der Heuristikwert errechnet sich auf Grundlage des gefundenen Zielzustands r.

Nun ist es möglich, dass in der Schicht m, in welcher r gefunden wurde, nicht nur ein r gewonnen wurde (d.h. $r \subseteq S[m]$), sondern gleich eine Menge R' an Zielzuständen (d.h. $r' \subseteq S[m]$ für alle $r' \in R'$). Da sich nicht notwendigerweise für alle $r' \in R'$ derselbe Heuristikwert ergeben muss, wird in der Implementierung der Rückwärtsschritt für alle $r' \in R'$ ausgeführt und der minimale Heuristikwert zurückgegeben.

3.2.3 Expansion suboptimaler Knoten

AO* expandiert ausschließlich Knoten, die sich in optimalen Teillösungsgraphen befinden (vergleiche Zeile 9 in AO*). Existiert kein solcher Knoten, wird ein noch nicht expandierter Knoten maximaler Tiefe expandiert, der zwar bereits erzeugt wurde, sich aber nicht in einem optimalen Teillösungsgraphen befindet. Hierzu ist es erforderlich, alle erzeugten Knoten in eine weitere Menge abzulegen und bei Bedarf (falls kein Knoten in einem optimalen Teillösungsgraph existiert) einen Knoten aus dieser Menge zu selektieren. Erst wenn diese Menge leer ist, kann der Suchvorgang abgebrochen werden. Diese Vorgehensweise ist wichtig, da ohne sie keine Korrektheit gewährleistet werden kann. Die Korrektheit wäre verletzt, wenn sich zwar eine Lösung herleiten ließe, sich diese aber aufgrund falscher Kostenschätzungen in einem suboptimalen Teillösungsgraphen befindet. Maßgeblich sind Zyklen dafür verantwortlich, dass der AO*-Algorithmus aufgrund minimaler Kostenschätzungen einen optimalen Teillösungsgraphen berechnet, der aber nicht zu einem Lösungsgraphen erweitert werden kann.

4 Ergebnisse

Der AO*-Algorithmus kann zu Testzwecken mit verschiedenen Einstellmöglichkeiten gestartet werden. Es kann gewählt werden, ob die normale FF-Heuristik verwendet wird, die erweiterte FF-Heuristik, oder ob uninformierte Suche vorgenommen wird. Die uninformierte Suche weist allen neuen Knoten (die keinen Zielzuständen entsprechen) den Heuristikwert 1 zu. Weiterhin kann eingestellt werden, ob die Kostenschätzung der UND-Knoten durch die Summation der Kind-Kostenschätzungen errechnet wird, oder durch deren Maximierung. Eine weitere Einstellmöglichkeit stellt die Selektionsstrategie der FF-Heuristik dar. Wir erinnern uns, dass die Funktion **selectAction(layer** i**, statevariable** g**)**) dafür verantwortlich ist, eine Regel zu selektieren, falls mehrere in Frage kommen. Implementiert wurden 3 Strategien: Selektion einer Regel, die die kleinste Precondition-List besitzt, Selektion einer Regel, die die größte Add-List besitzt und schließlich zufällige Selektion.
Folgend werden die Ergebnisse einiger Kombinationen dieser Einstellmöglichkeiten anhand von zwei Erreichbarkeitsspielen vorgestellt. Die Tests wurden auf einem Windows Vista x64 System durchgeführt mit einem Intel Core 2 Duo Prozessor (zwei Prozessorkerne) mit 2.4GHz und 2GB RAM.

4.1 Tic Tac Toe

Die Kodierung des Erreichbarkeitsspiels Tic Tac Toe wurde bereits in den Beispielen 3.1.1.1 und 3.1.2.1 vorgestellt. Tic Tac Toe zeichnet sich dadurch aus, dass keine Zyklen existieren und auch keine Gewinnstrategie für den Startspieler.

	Variante 1	Variante 2
Anzahl äußere Iterationen:	1052	1088
Verbrauchte Zeit:	00:00:452	00:00:503
Heuristik:	erweiterte FF	erweiterte FF
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	4715	4786
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	4715	4786
ODER-Spieler gewinnt?	nein	nein

	Variante 3	Variante 4
Anzahl äußere Iterationen:	1127	1108
Verbrauchte Zeit:	00:00:493	00:00:508
Heuristik:	normale FF	normale FF
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	4822	4808
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	4822	4808
ODER-Spieler gewinnt?	nein	nein

	Variante 5	Variante 6
Anzahl äußere Iterationen:	906	970
Verbrauchte Zeit:	00:00:335	00:00:359
Heuristik:	konstant 1	konstant 1
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	4330	4385
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	4330	4385
ODER-Spieler gewinnt?	nein	nein

Es folgt eine Zusammenfassung der wichtigsten Ergebnisse:

- Es kann in etwa einer halben Sekunde bewiesen werden, dass keine Gewinnstrategie für den Startspieler von Tic Tac Toe existiert.
- Am besten schneidet die uninformierte Suche ab, die deutlich weniger Knoten erstellt, als die Suche mit Heuristiken.
- Fast gleich viele Knoten erstellen die beiden Heuristiken, wobei die erweiterte FF-Heuristik etwas besser abschneidet als die normale.

4.2 Warentransport

Dieses Erreichbarkeitsspiel entstammt folgendem Szenario:
Zwei Piloten (Pilot, der ODER-Spieler, und Co-Pilot, der UND-Spieler) haben die Aufgabe, Waren aus drei Flughäfen neu zu verteilen. In jedem Flughafen sind zwei Waren deponiert. In Araxos lagern Wein und Honig, in Zürich lagern Schokolade und Milch und in Stuttgart lagern Maultaschen und das weltgrößte transportable Riesenrad. Araxos sollen die Maultaschen und die Schokolade geliefert werden, Zürich das Riesenrad und der Wein und schließlich soll Stuttgart mit dem Honig und der Milch beliefert werden. Hierzu stehen den beiden Piloten verschiedene Aktionen zur Verfügung:

Der Pilot kann

- von einem Flughafen zu einem weiteren fliegen, sofern der Tank voll ist,
- das Flugzeug mit einer einzigen Ware (pro Ausführung) beladen,
- aus dem Flugzeug eine einzige Ware (pro Ausführung) ausladen, sofern diese Ware nicht gerade eingeladen wurde,
- rauchen (no-Operation). Da er Raucher ist, darf er das Flugzeug nicht betanken.

Der Co-Pilot kann

- das Flugzeug betanken, sofern der Tank leer ist,
- aus dem Flugzeug eine einzige Ware (pro Ausführung) ausladen, sofern diese Ware nicht gerade eingeladen wurde,
- eine Kaffepause einlegen (no-Operation), sofern er dies nicht gerade getan hat.

Im Gegensatz zu Tic Tac Toe enthält dieses Erreichbarkeitsspiel neben Transpositionen auch Zyklen.

	Variante 1	Variante 2
Anzahl äußere Iterationen:	1307	46609
Verbrauchte Zeit:	00:02:208	01:08:778
Heuristik:	erweiterte FF	erweiterte FF
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	7957	189602
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	156	77
ODER-Spieler gewinnt?	ja	ja

	Variante 3	Variante 4
Anzahl äußere Iterationen:	10457	56443
Verbrauchte Zeit:	00:14:557	01:33:750
Heuristik:	normale FF	normale FF
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	62081	209227
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	165	80
ODER-Spieler gewinnt?	ja	ja

	Variante 5	Variante 6
Anzahl äußere Iterationen:	60987	55454
Verbrauchte Zeit:	00:31:505	00:40:090
Heuristik:	konstant 1	konstant 1
FF-Selektionsvariante:	kl. Precondition-List	kl. Precondition-List
UND-Kostenschätzung:	Maximum	Summe
Anzahl ersteller Knoten:	235172	215600
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	174	131
ODER-Spieler gewinnt?	ja	ja

Es folgt eine Zusammenfassung der wichtigsten Ergebnisse:

- Die Summen-Kostenschätzung führt wie in 2.2.2 (Kostenschätzung der UND-Knoten) prophezeit, stets zu kleineren Lösungsgraphen als die Maximum-Kostenschätzung (20% bis 50%). Dies geht allerdings auf Kosten der Laufzeit und des Speicherbedarfs.
- Die konstante Heuristik schneidet am schlechtesten ab. Sie expandiert die meisten Knoten und führt zu den größten Lösungsgraphen.
- Die erweiterte FF-Heuristik ist der normalen in allen Punkten überlegen. Im Falle der Summen-Kostenschätzung findet sie einen etwas kleineren Lösungsgraphen. Weiterhin werden etwa 10% weniger Knoten expandiert, und auch die Laufzeit ist etwa 25% geringer. Die größten Unterschiede ergeben sich allerdings bei Verwendung der Maximum-Kostenschätzung. Zwar ist der gefundene Lösungsgraph auch hier nur minimalst kleiner, doch beträgt die Laufzeit lediglich ein Siebtel im Vergleich zur Verwendung der normalen FF-Heuristik. Weiterhin werden knapp 90% weniger Knoten expandiert.

Zusammenfassend kann gesagt werden, dass die erweiterte FF-Heuristik den anderen Heuristiken in allen Punkten mindestens leicht überlegen ist.

Betrachten wir nun zwei weitere Selektionsstrategien der FF-Heuristik. Die Selektionsstrategie, die stets eine Regel selektiert, welche die größte Add-List besitzt, erzielt exakt dieselben Ergebnisse wie die Strategie, die stets eine Regel mit kleinster Precondition-List selektiert. Dies liegt allerdings nur daran, dass in diesem Erreichbarkeitsspiel alle Regeln mit kleinster Precondition-List gleichzeitig die größte Add-List besitzen. Die letzte implementierte Selektionsstrategie ist das zufällige Selektieren von Regeln. Um diese Selektionsstrategie zu untersuchen, beschränken wir uns auf Verwendung der erweiterten FF-Heuristik. Die folgenden Ergebnisse entstammen zehn Testdurchläufen.

	bestes Ergebnis	schlechtestes Ergebnis
Anzahl äußere Iterationen:	48720	44000
Verbrauchte Zeit:	01:18:860	01:11:443
Heuristik:	erweiterte FF	erweiterte FF
FF-Selektionsvariante:	zufällige Selektion	zufällige Selektion
UND-Kostenschätzung:	Summe	Summe
Anzahl ersteller Knoten:	196509	184490
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	73	100
ODER-Spieler gewinnt?	ja	ja

Diese beiden Ergebnisse zeigen das beste und schlechteste Ergebnis bezüglich Größe des Lösungsgraphen unter Verwendung der Summen-Kostenschätzung. In keinem der zehn Testdurchläufe kam es zu einem Überlauf. Bei Betrachtung des besten Ergebnisses erkennt man, dass die randomisierte Variante sogar einen etwas kleineren Lösungsgraphen findet als die nichtrandomisierte. Dafür benötigt sie geringfügig mehr Zeit und expandiert etwas mehr Knoten. Das schlechteste Ergebnis berechnet bei sonst ähnlich großen Werten einen etwa 25% größeren Lösungsgraphen.

Betrachten wir nun dasselbe für die Maximum-Kostenschätzung.

	bestes Ergebnis	schlechtestes Ergebnis
Anzahl äußere Iterationen:	2767	5613
Verbrauchte Zeit:	00:04:551	00:09:261
Heuristik:	erweiterte FF	erweiterte FF
FF-Selektionsvariante:	zufällige Selektion	zufällige Selektion
UND-Kostenschätzung:	Maximum	Maximum
Anzahl ersteller Knoten:	15411	31735
Anzahl Überläufe:	0	0
Anzahl Knoten im Lösungsgraphen:	158	286
ODER-Spieler gewinnt?	ja	ja

Diese beiden Ergebnisse zeigen das beste und schlechteste Ergebnis bezüglich der Größe des Lösungsgraphen unter Verwendung der Maximum-Kostenschätzung. Betrachtet man das beste Ergebnis, so erkennt man, dass der Lösungsgraph minimal größer ist als in der nichtrandomisierten Variante. Weiterhin wird etwa doppelt so viel Zeit benötigt und es werden etwa doppelt so viele Knoten expandiert. Das schlechteste Ergebnis weist in allen Kriterien nochmals deutlich schlechtere Ergebnisse auf.

Insgesamt kann damit gesagt werden, dass die präferierte Strategie (Selektion einer Regel mit kleinster Precondition-List) gute und verlässliche Ergebnisse liefert und damit eine sinnvolle Wahl darstellt.

Literaturverzeichnis

[1] Allis, L. V., M. van der Meulen und H. J. van den Herik: *Proof-Number Search*. Artificial Intelligence, 66(1):91–124, 1994.

[2] Bonet, B. und H. Geffner: *HSP: Heuristic search planner*. AIPS-98 Planning Competition, Pittsburgh, PA, 1998.

[3] Bylander, T.: *The Computational Complexity of Propositional STRIPS Planning*. Artificial Intelligence, 69(1-2):165–204, 1994.

[4] Cocosco, C. A.: *A Review of "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving by R.E. Fikes, N.J. Nillson, 1971"*, 1981.

[5] de Alfaro, L., T. A. Henzinger und O. Kupferman: *Concurrent Reachability Games*. Theoretical Computer Science, 386(3):188–217, 1998.

[6] Fikes, R. E. und N. J. Nilsson: *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artificial Intelligence, 2(3/4):189–208, 1971.

[7] Grädel, E., W. Thomas und T. Wilke: *Automata, Logics, and Infinite Games. A Guide to Current Research*, Kap. 2, S. 23–40. Springer-Verlag, 2002.

[8] Hansen, E. A. und S. Zilberstein: *Heuristic Search in Cyclic AND/OR Graphs*. Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), S. 412–418, 1998.

[9] Hart, P. E., N. J. Nilsson und B. Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, 1968.

[10] Hart, P. E., N. J. Nilsson und B. Raphael: *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*. SIGART Newsletter, 37:28–29, 1972.

[11] Helmert, M., R. Mattmüller und S. Schewe: *Selective Approaches for Solving Weak Games*. In: *Proceedings of the Fourth International Symposium on Automated Technology for Verification and Analysis (ATVA 2006)*, Nr. 4218 in *Lecture Notes in Computer Science (LNCS)*, S. 200–214, Berlin / Heidelberg, 2006. Springer-Verlag.

[12] Hoffmann, J.: *FF: The Fast-Forward Planning System*. AI Magazine, 22(3):57–62, 2001.

[13] Hoffmann, J. und B. Nebel: *The FF Planning System: The Fast Plan Generation Through Heuristic Search*. Journal of Artificial Intelligence Research, 14:253–302, 2001.

[14] Nilsson, N. J.: *Principles of Artificial Intelligence*, Kap. 3, S. 99–129. Springer-Verlag, 1998.

[15] Schijf, M.: *Proof-Number Search and Transpositions*. Masterarbeit, University of Leiden, 1993.