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

Albert-Ludwigs-Universität
Freiburg im Breisgau

Diplomarbeit

Anwendung von Pattern-Database-Heuristiken zum Lösen nichtdeterministischer Planungsprobleme

---

Autor: Pascal Bercher
Betreuer: Robert Mattmüller

Erstgutachter: Prof. Dr. Bernhard Nebel
Zweitgutachter: Prof. Dr. Christoph Scholl

Datum: 11. März 2009
(überarbeitet: 14. Mai 2009)

Zusammenfassung

Die vorliegende Arbeit behandelt das Suchen von starken Plänen nichtdeterministischer Planungsprobleme mit dem Ansatz des Planens durch heuristische Suche unter Verwendung des AO*-Algorithmus und von domänenunabhängigen Pattern-Database-Heuristiken. Starke Pläne garantieren das Erreichen eines Zielzustands unabhängig vom Ausgang des Nichtdeterminismus. Der Fokus dieser Arbeit richtet sich auf die Pattern-Database-Heuristiken, die im klassischen Planen Anwendung finden und nun auf nichtdeterministisches Planen übertragen werden. Es wird gezeigt, unter welchen Bedingungen man durch Addition von Pattern-Database-Heuristiken eine zulässige und informative Heuristik konstruieren kann. Der Suchalgorithmus und die Heuristik wurden vollständig implementiert. Anhand dieser Implementierung wurden drei Domänen untersucht. Die Ergebnisse dieser Experimente werden schließlich diskutiert. Dabei zeigt sich, dass Pattern-Database-Heuristiken, insbesondere durch Ausnutzung von Additivität, auch im nichtdeterministischen Planen sehr gute Ergebnisse erzielen können.

Danksagungen

Zunächst möchte ich mich bei Herrn Prof. Dr. Bernhard Nebel dafür bedanken, meine Diplomarbeit an seinem Lehrstuhl anfertigen zu dürfen. Weiterhin bedanke ich mich bei ihm und bei Herrn Prof. Dr. Scholl für ihre Bereitschaft, sich als Gutachter zur Verfügung zu stellen.

Einen ganz besonderen Dank möchte ich Herrn Robert Mattmüller für die hervorragende Betreuung dieser Arbeit aussprechen. Er war stets dazu bereit, die Flut meiner Emails ausführlich zu beantworten, sehr oft binnen weniger Minuten. Die vielen Diskussionen und unzähligen Treffen haben mich weiterhin sehr bei dieser Arbeit unterstützt und trugen letztlich maßgeblich zu ihrem Gelingen bei.

Ein weiterer Dank gilt Marina Klingele, Sarah Schwarzkopf, Christoph Betz und Michael Meier für das Korrekturlesen dieser Arbeit und die vielen Verbesserungsvorschläge.

Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mir das Studium der Informatik erst ermöglicht haben und ohne deren Unterstützung auch diese Arbeit nicht möglich gewesen wäre.

Erklärung

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

Freiburg im Breisgau, 11. März 2009 _____

(Pascal Bercher)

Inhaltsverzeichnis

1. Einleitung **1**

2. Nichtdeterministische Planungsprobleme **3**

2.1. Grundlagen 3

2.2. Beispiel eines nichtdeterministischen Planungsproblems 6

3. Die Suche nach starken Plänen mit dem AO*-Algorithmus **11**

3.1. Der Suchalgorithmus AO* 11

3.2. Beispiel zum AO*-Algorithmus 13

4. Pattern-Database-Heuristiken **17**

4.1. Grundlagen 18

4.2. Beispiel zur Abstraktion mit Patterns 21

4.3. Additivität 24

4.3.1. Beispiel für additive Patterns 32

4.3.2. Additivität in der Praxis 34

4.4. Berechnung und Speicherung der Pattern-Database-Heuristiken 36

5. Implementierung **41**

5.1. Vorverarbeitung 41

5.2. Terminierung der Value-Iteration 41

6. Experimente **45**

6.1. Tireworld 47

6.1.1. Problembeschreibung 47

6.1.2. Beispiel 48

6.1.3. Diskussion der Ergebnisse 48

6.2. Chain-of-Rooms 53

6.2.1. Problembeschreibung 53

6.2.2. Beispiel 53

6.2.3. Diskussion der Ergebnisse 54
6.3. Coin-Flip 61
6.3.1. Problembeschreibung 61
6.3.2. Beispiel 61
6.3.3. Diskussion der Ergebnisse 61

7. Zusammenfassung und Ausblick **67**
7.1. Zusammenfassung 67
7.2. Ausblick 68

Literaturverzeichnis **69**

A. Abbildungsverzeichnis **73**

B. Formale Problembeschreibungen **75**
B.1. Tireworld-Kodierungen 75
B.2. Chain-of-Rooms-Kodierungen 77
B.3. Coin-Flip-Kodierungen 79

1. Einleitung

Handlungsplanung ist eines der Hauptforschungsgebiete der künstlichen Intelligenz und beschäftigt sich mit dem Lösen von Planungsproblemen, die sich dadurch auszeichnen, dass ein oder mehrere Initialzustände mit Hilfe von Aktionen in einen beliebigen Zielzustand überführt werden müssen. Im *klassischen Planen* beschränkt man sich auf deterministische Aktionen, also auf solche, deren Ausgang eindeutig bestimmt ist. Weiterhin gilt die Annahme der Endlichkeit, das heißt es existieren nur endlich viele erreichbare Zustände, sowie die Annahme der vollen Observierbarkeit, das heißt in jedem Zustand sind stets alle Fakten bekannt. In der vorliegenden Arbeit werden nichtdeterministische Planungsprobleme betrachtet, welche sich von Problemen des klassischen Planens durch den nichtdeterministischen Ausgang von Aktionen unterscheiden.
Es existieren verschiedene Ansätze zum Lösen von Planungsproblemen. Einige der populärsten Ansätze sind *SAT-Planen* [23, 24], bzw. eine Erweiterung dessen auf *QBF-Planen* [30], *Planen durch Model-Checking* [7], entscheidungstheoretisches Planen (MDPs) [34] und *Planen durch heuristische Suche* [4, 22, 19]. Während die Repräsentation der Welt beim SAT- und QBF-Planen durch (quantifizierte) aussagenlogische Formeln vorgenommen wird, wird sie beim Planen durch Model-Checking üblicherweise symbolisch repräsentiert, zum Beispiel durch BDDs [5]. Im Gegensatz hierzu werden beim Planen durch heuristische Suche die Zustände üblicherweise explizit repräsentiert, das heißt durch Angabe aller Fakten, die im jeweiligen Zustand wahr sind. Pläne werden durch Suche in einem Zustandsraum generiert, der durch Expansion von Zuständen (das heißt der Anwendung von Aktionen) sukzessive erweitert wird. Im Allgemeinen stehen stets mehrere Zustände zur Expansion zur Auswahl. Die Wahl des Zustands zur Expansion ist maßgeblich für die Qualität der Lösung und die Terminierungsgeschwindigkeit verantwortlich. Die Wahl dieses Zustands wird im Planen durch heuristische Suche durch die Verwendung einer Heuristik geleitet, die schätzt, wie weit dieser Zustand noch von einem Zielzustand entfernt ist. Für konkrete Planungsprobleme ist es oftmals möglich, äußerst informative Heuristiken von Hand zu gewinnen, da domänenspezifisches Wissen eingebracht werden kann. Das Entwickeln von aussagekräftigen, domänenunabhängigen Heuristiken ist aufgrund ihrer Allgemeinheit deutlich schwieriger.
Im klassischen Planen durch heuristische Suche konnten in den letzten Jahren sehr große Fortschritte erzielt werden. Die größten Erfolge konnten dabei unter anderem durch die Anwendung von Pattern-Database-Heuristiken erzielt werden. Pattern-Database-Heuristiken zeichnen sich insbesondere dadurch aus, dass vor der eigentlichen Suche im Zustandsraum das Planungsproblem durch Abstraktion vereinfacht und optimal gelöst wird. Alle in diesem abstrakten Zustandsraum er-

reichbaren Zustände werden zusammen mit ihrem optimalen Kostenwert in einer Pattern-Database abgelegt. Bei Erreichen eines Zustands s während des Suchvorgangs wird auf den Zustand s' zugegriffen, welcher der Abstraktion von s entspricht, um dessen optimale Kosten der Abstraktion für den Heuristikwert zu verwenden. Die vorliegende Arbeit adaptiert die Idee der Pattern-Database-Heuristiken des klassischen Planens und überträgt sie auf nichtdeterministische Planungsprobleme, bei denen Aktionen potentiell mehrere Ausgänge haben können.
Während man von Hand beliebig konstruierte, nichtdeterministische Planungsprobleme definieren kann, ist der Nichtdeterminismus in der Praxis häufig durch Unzuverlässigkeit von Sensoren und/oder Aktuatoren gegeben. In beiden Fällen setzen sich die nichtdeterministischen Effekte, neben dem entsprechenden gewünschten Effekt (die gemessenen Daten sind korrekt, bzw. erfolgreiche Aktionsausführung), aus den unerwünschten Nebeneffekten (die gemessenen Daten sind falsch, bzw. misslungene Aktionsausführung) zusammen. In solchen Fällen ist neben dem Lösen von nichtdeterministischen Planungsproblemen auch der alternative Ansatz des *Replanning* möglich. Dabei wird zunächst ein deterministischer Plan gesucht, davon ausgehend, dass alle Sensordaten korrekt sind und alle Aktionen gelingen. Erst im Fall, dass sich ein Sensordatum als falsch erweist oder eine Aktion misslingt, wird ein neuer Plan berechnet.

Es folgt eine eine kurze Übersicht, welche Themen in den folgenden Kapiteln behandelt werden:

Kapitel 2 ist das Grundlagenkapitel dieser Arbeit; dort wird formal definiert, was ein nichtdeterministisches Planungsproblem ist und was man unter einer Lösung eines solchen versteht. Die Suche nach einer Lösung eines nichtdeterministischen Planungsproblems erfolgt mit dem Ansatz *Planen durch heuristische Suche*. Der hierfür verwendete AO*-Suchalgorithmus wird in Kapitel 3 in Kürze erörtert und an einem Beispiel illustriert. Der Schwerpunkt dieser Arbeit stellt Kapitel 4 dar, welches die Pattern-Database-Heuristiken diskutiert, die vom AO*-Algorithmus verwendet werden, um die Suche zu lenken. Das Hauptergebnis dieses Kapitels wird ein Satz sein, der im klassischen Planen bereits gezeigt ist und der auf nichtdeterministisches Planen übertragen wird. Er zeigt Voraussetzungen auf, unter welchen man durch Addition verschiedener Heuristikwerte eine informativere sowie zulässige Heuristik erhält. Kapitel 5 geht kurz auf eine technische Feinheit der Implementierung ein, während Kapitel 6 die Ergebnisse der Implementierung anhand dreier Beispieldomänen diskutiert. Kapitel 7 schließt diese Arbeit mit einer Zusammenfassung und einer Diskussion über mögliche Verbesserungen und Erweiterungen ab.

2. Nichtdeterministische Planungsprobleme

2.1. Grundlagen

Es werden nichtdeterministische Planungsprobleme mit vollständiger Information, das heißt mit voller Observierbarkeit betrachtet. Ein Agent besitzt hierzu eine endliche Menge von Aktionen, welche nichtdeterministisch zu verschiedenen Ausgängen führen. Die Aufgabe des Agenten besteht darin, einen gegebenen Initialzustand in einen beliebigen Zielzustand zu überführen, unabhängig von den nichtdeterministischen Ausgängen der Aktionen, die ihm hierfür zur Verfügung stehen. Um diese Aufgabe zu präzisieren, folgen zunächst einige Definitionen.

Die Definitionen 2.1, 2.2, 2.3, 2.6, 2.7 und 2.8 sind teilweise sehr stark angelehnt an die entsprechenden Definitionen aus der Studienarbeit [2], auf der diese Arbeit aufbaut.

Definition 2.1 (Nichtdeterministisches Planungsproblem)**.**
Ein nichtdeterministisches Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ besteht aus den folgenden Komponenten:

- Einer endlichen Menge S von Zuständen.
- Einem Initialzustand $s_0 \in S$.
- Einer endlichen Menge A von Aktionen.
- Einer Zugzuweisungsfunktion $\Gamma : S \to 2^A$, die jedem Zustand $s \in S$ die Menge der in s anwendbaren Aktionen zuweist. Es wird für alle $s \in G$ gefordert, dass $\Gamma(s) = \emptyset$.
- Einer partiellen Transitionsfunktion $\delta : S \times A \to 2^S$, die jedem Zustand $s \in S$ und jeder Aktion $a \in \Gamma(s)$ die Menge der möglichen Nachfolgezustände zuweist. δ ist partiell, da sie insbesondere auf solchen Tupeln (s, a) nicht definiert ist, für die $a \notin \Gamma(s)$.
- Einer nichtleeren Menge $G \subseteq S$ von Zielzuständen.

Definition 2.2 (Strategie)**.**
Eine Strategie ist eine partielle Abbildung $\pi : S \to A$, die einen Zustand $s \in S$ auf eine in s anwendbare Aktion abbildet. Das heißt, falls für $s \in S$ und $a \in A$, $\pi(s) = a$ gilt, dann gilt auch $a \in \Gamma(s)$. π ist partiell, da sie nicht zwangsläufig auf allen $s \in S$ definiert sein

muss, sondern lediglich auf den Zuständen, die von s_0 ausgehend durch die Strategie π erreichbar sind.

Definition 2.3 (Historie, endliche Historie)**.**
Eine durch eine Strategie π induzierte Historie σ_π von $\mathcal{P}$ ist eine Folge von Zuständen, die durch Spielen von π möglicherweise auftreten kann. σ_π ist damit entweder eine eine endliche Folge $(s_0, \ldots, s_n)$ mit $n \in \mathbb{N} \cup \{0\}$ oder eine unendliche Folge $(s_0, s_1, \ldots)$, für die gilt:

- s_0 ist Initialzustand von $\mathcal{P}$ und
- für alle $i \geq 0$ gilt: Falls $\pi(s_i)$ nicht definiert ist, endet die Folge auf s_i. Sonst gilt $s_{i+1} \in \delta(s_i, \pi(s_i))$.

Durch das Auftreten von Zyklen (das heißt, falls für den Präfix $(s_0, \ldots, s_j)$ mit $j \in \mathbb{N} \cup \{0\}$ einer Historie ein $s_i \in S$ und ein $a \in \Gamma(s_j)$ existiert mit $s_i \in \delta(s_j, a)$ und $i \leq j$) kann eine Historie unendlich werden.
Eine endliche Historie ist eine Historie $(s_0, \ldots, s_n)$ mit $n \in \mathbb{N} \cup \{0\}$.

Die folgenden Definitionen der Begriffe schwacher, starker und starker zyklischer Plan wurden von Cimatti u.a. [7] in einer äquivalenten Formulierung eingeführt.

Definition 2.4 (Schwacher Plan, starker Plan, starker zyklischer Plan)**.**
Eine Strategie π heißt schwacher Plan, falls eine durch π induzierte Historie $\sigma_\pi = (s_0, \ldots, s_n)$ mit $n \in \mathbb{N} \cup \{0\}$ existiert, so dass $s_n \in G$. Durch Ausführung eines schwachen Plans kann daher ein Zielzustand nach endlicher Zeit erreicht werden, jedoch nicht zwingend.
Eine Strategie π heißt starker Plan, falls alle durch π induzierten Historien endlich sind und falls für jede solche Historie $\sigma_\pi = (s_0, \ldots, s_n)$ mit $n \in \mathbb{N} \cup \{0\}$ gilt, dass $s_n \in G$. Ein starker Plan gewährleistet damit das Erreichen eines Zielzustands in endlicher Zeit, unabhängig vom Nichtdeterminismus.
Eine Strategie π heißt starker zyklischer Plan, falls für alle endlichen, durch π induzierten Historien $\sigma_\pi = (s_0, \ldots, s_n)$ mit $n \in \mathbb{N} \cup \{0\}$ gilt, dass $s_n \in G$ und für alle unendlichen, durch π induzierten Historien $\sigma_\pi = (s_0, s_1, \ldots)$ gilt, dass für alle $i \in \mathbb{N} \cup \{0\}$, $(s_0, \ldots, s_i)$ Präfix einer endlichen, durch π induzierten Historie $(s_0, \ldots, s_i, \ldots, s_{n'})$, $n' \in \mathbb{N} \cup \{0\}$ mit $s_{n'} \in G$, ist. Starke zyklische Pläne können folglich in möglicherweise unendlich lange Folgen von Zuständen resultieren, doch muss aus jedem Zustand jeder Historie in endlich vielen Schritten ein Zielzustand erreichbar sein.

Es soll eine Strategie gefunden werden, die in endlicher Zeit, unabhängig vom Ausgang des Nichtdeterminismus, einen Zielzustand herleitet. Eine solche Strategie entspricht einem starken Plan. Ein Zustand $s \in S$ heißt gewonnen, falls für diesen ein starker Plan existiert; sonst heißt er verloren. Im Kontext von nichtdeterministischen Planungsproblemen sind außerdem konformante Pläne erwähnenswert. Da das Finden konformanter Pläne nicht Gegenstand dieser Arbeit ist, wird deren Definition lediglich informell und zwecks Vollständigkeit angegeben.

Definition 2.5 (Konformanter Plan)**.**
Ein konformanter Plan ist eine endliche Sequenz von Aktionen $(a_1, \ldots, a_n)$ mit $n \in \mathbb{N} \cup \{0\}$, die angewendet auf den Initialzustand s_0 unabhängig vom Nichtdeterminismus das Erreichen eines Zielzustands garantiert. Die Sequenz $(a_1, \ldots, a_n)$ mit $n = 0$ entspricht der leeren Sequenz.

Diese Arbeit beschränkt sich auf das Finden von starken Plänen. Die im nächsten Kapitel vorgestellten Techniken der Pattern-Database-Heuristiken wären allerdings auch für die Suche nach schwachen oder starken zyklischen Plänen geeignet, ohne dass gravierende Anpassungen am Ansatz der Pattern-Database-Heuristiken notwendig würden. Um auch starke zyklische Pläne finden zu können, wäre zunächst eine Anpassung des Suchalgorithmus erforderlich. Welche Anpassungen bei der Berechnung der Pattern-Database-Heuristiken erforderlich wären, wird in dieser Arbeit nicht näher erörtert.
Starke zyklische Pläne sind (je nach Domäne) genauso sinnvoll wie starke Pläne, wie das folgende einfache Beispiel zeigt. Es sei die Aufgabe gegeben, mit einem Würfel mit beliebig vielen Versuchen eine 6 zu würfeln. Es existiert kein starker Plan, da es eine unendlich lange Historie gibt, in der niemals eine 6 geworfen wird. Es existiert aber ein starker zyklischer Plan, da stets die 6 geworfen werden *könnte*. Der starke zyklische Plan entspräche daher dem wiederholten Würfeln, bis die 6 geworfen wird.

Ein nichtdeterministisches Planungsproblem $\mathcal{P}$ kann eindeutig durch einen endlichen UND/ODER-Graphen repräsentiert werden.

Definition 2.6 (UND/ODER-Graph)**.**
Ein UND/ODER-Graph $\mathcal{G} = (V, C)$ besteht aus einer endlichen Menge V von Knoten und einer endlichen Menge von Konnektoren C, wobei jeder Konnektor $c \in C$ aus einem Vorgängerknoten $v \in V$ und beliebig vielen Nachfolgerknoten $v' \in V$ besteht, das heißt ein Konnektor hat die Form $c = (v, \{v_1, \ldots, v_n\})$, mit $v \in V$ Vorgänger und alle $v' \in \{v_1, \ldots, v_n\} \subseteq V$ für $n \in \mathbb{N}$ sind Nachfolger von v. Für $c = (v, \{v_1, \ldots, v_n\})$ wird der Knoten v als Vorgänger von c, $pred(c)$, bezeichnet, und die Menge $\{v_1, \ldots, v_n\}$ als Nachfolger von c, $succ(c)$. Durch $C(v)$ für $v \in V$ wird die Menge aller Konnektoren gekennzeichnet, deren Vorgängerknoten v ist, das heißt $C(v) := \{\, c \in C \mid pred(c) = v \,\}$.

Definition 2.7 (Vollständiger UND/ODER-Graph von $\mathcal{P}$)**.**
Sei $\mathcal{G}' = (V', C')$ mit $V' = S$ und $C' = \{\, (s, \delta(s, a)) \mid s \in S \text{ und } a \in \Gamma(s) \,\}$. Der vollständige UND/ODER-Graph eines nichtdeterministischen Planungsproblems $\mathcal{P}$ ist nun die Zusammenhangskomponente $\mathcal{G}$ von $\mathcal{G}'$, die s_0 enthält.

Das nächste Kapitel behandelt die Suche nach starken Plänen. Diese Pläne werden durch so genannte Lösungsgraphen repräsentiert, welche eine direkte Kodierung der gesuchten starken Pläne darstellen. Dabei wird jeder durch diese Strategie erreichbare Zustand in den Graphen übernommen, so dass alle Blätter dieses Graphen Zielzustände sind und keine Zykel auftreten.

Definition 2.8 (Lösungsgraph von $\mathcal{P}$)**.**
Ein Lösungsgraph von $\mathcal{P}$ ist ein zusammenhängender zyklenfreier UND/ODER-Graph $\mathcal{G} = (V, C)$ mit Wurzelknoten s_0, der Teilgraph des vollständigen UND/ODER-Graphen (V', C') von $\mathcal{P}$ ist, so dass alle inneren Knoten genau einen ausgehenden Konnektor besitzen und alle Blätter von $\mathcal{G}$ Zielzustände sind, das heißt für alle $v \in V$ mit $C(v) = \emptyset$ gilt $v \in G$. Ein optimaler Lösungsgraph ist ein Lösungsgraph mit minimaler Tiefe. Da Lösungsgraphen zyklenfrei sind, ist ihre Tiefe endlich.

Da jeder Lösungsgraph genau einem starken Plan entspricht, existiert für einen Zustand $s \in S$ genau dann ein starker Plan, wenn ein Lösungsgraph mit Wurzel s existiert. Diesen Zusammenhang kann man formal auch über die Kosten eines UND/ODER-Graphen ausdrücken.

Definition 2.9 (Kosten eines vollständigen UND/ODER-Graphen von $\mathcal{P}$)**.**

$$c_0^*(s) = 0 \quad \text{für alle } s \in S$$

$$c_{i+1}^*(s) = \begin{cases} 0 & \text{falls } s \in G \\ \infty & \text{falls } C(s) = \emptyset \text{ und } s \notin G \\ \min\limits_{c \in C(s)} \max\limits_{s' \in succ(c)} (c_i^*(s') + 1) & \text{sonst} \end{cases}$$

Die Kosten eines vollständigen UND/ODER-Graphen (V, C) von $\mathcal{P}$ mit Wurzel s wird definiert als $c^*(s) = \lim_{i \to \infty} c_i^*(s)$. Der Limes der Folge $(c^*(s)_i)_{i \in \mathbb{N} \cup \{0\}}$ existiert, da die Folge aufgrund ihrer Monotonie entweder *konvergiert* oder *bestimmt divergiert*.

Die Kosten $c^*(s)$ eines vollständigen UND/ODER-Graphen ergeben ∞, falls kein Lösungsgraph existiert und entsprechen anderenfalls, wegen der Minimierung über die Konnektoren, der Tiefe eines optimalen Lösungsgraphen von $\mathcal{P}$. Daher wird c^* auch als die *optimalen Kosten* bezeichnet. Nun kann ausgesagt werden, dass für einen Zustand $s \in S$ genau dann ein starker Plan existiert, wenn $c^*(s) < \infty$ gilt.
Die Kosten eines Lösungsgraphen entsprechen seiner Tiefe und damit der Anzahl der Aktionen, die im schlechtesten Fall (je nach Ausgang des Nichtdeterminismus) ausgeführt werden müssen, um den Initialzustand in einen Zielzustand zu überführen.

2.2. Beispiel eines nichtdeterministischen Planungsproblems

Betrachten wir das nichtdeterministische Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ mit

- $S = \{s_0, \ldots, s_{13}\}$,
- $A = \{a_0, \ldots, a_6\}$,
- $\Gamma(s) = \emptyset$ für alle $s \in S \backslash \{s_0, s_3, s_4, s_6\}$ und

  $\Gamma(s_0) = \{a_0, a_1\} \quad \Gamma(s_3) = \{a_2, a_3\}$

$\Gamma(s_4) = \{a_4, a_5\} \quad \Gamma(s_6) = \{a_6\}$

- $\delta : S \times A \to 2^S$ ist gegeben durch:

$\delta(s_0, a_0) = \{s_1, s_2\} \quad \delta(s_3, a_2) = \{s_5, s_6\}$
$\delta(s_0, a_1) = \{s_3, s_4\} \quad \delta(s_3, a_3) = \{s_7, s_8, s_9\}$
$\delta(s_4, a_4) = \{s_9, s_{10}\} \quad \delta(s_6, a_6) = \{s_{12}, s_{13}\}$
$\delta(s_4, a_5) = \{s_4, s_{11}\}$

- $G = S \backslash \{s_0, s_2, s_3, s_4, s_6\}$.

Abbildung 2.1 zeigt den vollständigen UND/ODER-Graphen des Problems $\mathcal{P}$. In diesem Fall entspricht jede Aktion genau einem Konnektor und umgekehrt. Im Allgemeinen ist es jedoch möglich, dass ein Konnektor in einem UND/ODER-Graphen mehr als nur einer Aktion entspricht. Dies ist der Fall, falls in einem Zustand mehrere Aktionen dieselben Nachfolgezustände erzeugen. Konnektoren sind daran zu erkennen, dass alle Kanten, die zu demselben Konnektor gehören, durch einen Kreisausschnitt miteinander verbunden sind. Zielzustände sind durch einen sie zusätzlich umgebenden Kreis gekennzeichnet.

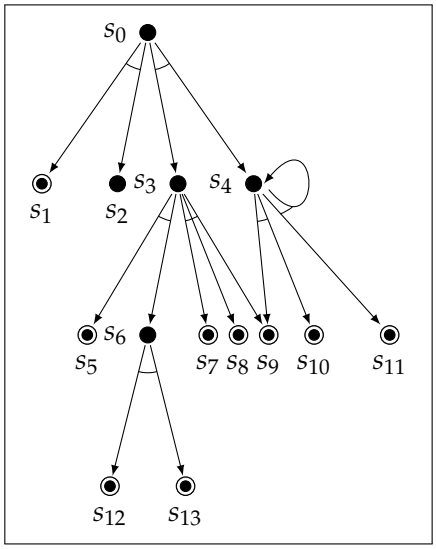
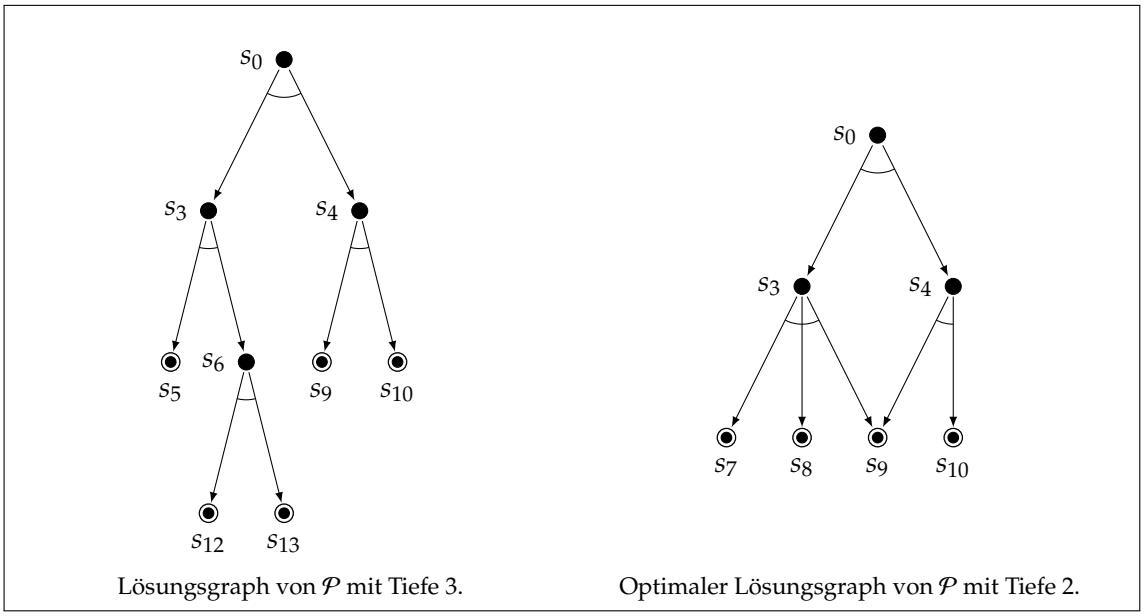


Abbildung 2.1.: Vollständiger UND/ODER-Graph von $\mathcal{P}$.

In diesem Beispiel existieren genau zwei Lösungsgraphen, welche in Abbildung 2.2 dargestellt sind. Der linke Lösungsgraph besitzt Tiefe 3 und damit Kosten 3, während der rechte Lösungsgraph Tiefe und Kosten 2 besitzt. Der zweite Lösungsgraph ist aufgrund seiner minimalen Tiefe ein optimaler Lösungsgraph, der in diesem Fall eindeutig ist, da kein weiterer Lösungsgraph mit Kosten 2 existiert.



Lösungsgraph von $\mathcal{P}$ mit Tiefe 3. Optimaler Lösungsgraph von $\mathcal{P}$ mit Tiefe 2.

Abbildung 2.2.: Sämtliche Lösungsgraphen von $\mathcal{P}$.

Betrachten wir nun exemplarisch, wieso es sich bei diesen beiden Graphen um Lösungsgraphen handelt und bei anderen nicht. Keiner der Teilgraphen der Lösungsgraphen kann Lösungsgraph sein, da sonst nicht alle Blätter Zielzustände wären. Aus demselben Grund ist der Graph, der lediglich aus den Knoten s_0, s_1, s_2 und aus dem Konnektor $(s_0, \{s_1, s_2\})$ besteht, kein Lösungsgraph (da s_2 zwar ein Blatt ist, aber kein Zielzustand).
Ein etwas interessanterer Fall ist der Graph, der sich aus dem rechten Lösungsgraphen ergibt, wenn man statt des Konnektors $(s_4, \{s_9, s_{10}\})$ den Konnektor $(s_4, \{s_4, s_{11}\})$ übernimmt (Abbildung 2.3). Obwohl dieser Graph nur Zielzustände als Blätter besitzt, handelt es sich um keinen Lösungsgraphen, da er nicht zyklenfrei ist. Solche Lösungen sollen ausgeschlossen werden, da der Nichtdeterminismus in diesem Fall dazu führen kann, dass der Zustand s_{11} niemals erreicht wird. Die zu Abbildung 2.3 korrespondierende Strategie würde einem starken *zyklischen* Plan entsprechen.

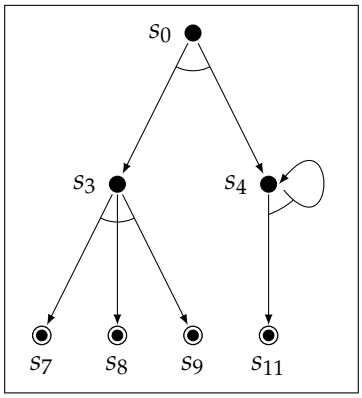


Abbildung 2.3.: Dieser UND/ODER-Graph von $\mathcal{P}$ stellt keinen Lösungsgraphen dar.

3. Die Suche nach starken Plänen mit dem AO*-Algorithmus

Die Suche nach einem starken Plan erfolgt durch den sukzessiven Aufbau eines UND/ODER-Graphen von $\mathcal{P}$. In diesem wird nach einem Lösungsgraphen gesucht. Hierzu dient eine Adaption des AO*-Algorithmus [28]. Während die Originalversion nur auf azyklischen Graphen arbeitet, ist die hier verwendete Adaption nicht auf diese Einschränkung angewiesen.

3.1. Der Suchalgorithmus AO*

Der hier vorgestellte Pseudocode (Algorithmus 3.1) stellt im Wesentlichen eine Übersetzung der Originalversion [28] dar, welche nicht mit Zyklen umgehen kann. Eine ausführliche Beschreibung der verwendeten Adaption findet sich in der Studienarbeit [2], aus welcher die für die Experimente verwendete Implementierung des Suchalgorithmus hervorging.

Der AO*-Algorithmus besteht aus zwei bedeutenden, sich abwechselnden Schritten, einem Vorwärtsschritt, welcher den UND/ODER-Graphen weiter expandiert, und einem Rückwärtsschritt, welcher die hierdurch gewonnenen Informationen weiter propagiert.

Der Vorwärtsschritt ist dafür verantwortlich, einen geeigneten Knoten zur Expansion auszuwählen, um diesen schließlich zu expandieren. Hierzu wird zunächst der so genannte *optimale Teillösungsgraph* $\mathcal{G}'$ bestimmt, welcher einem zusammenhängenden Teilgraphen des explizit repräsentierten Graphen $\mathcal{G}$ mit Wurzel s_0 entspricht, der lediglich solche Konnektoren enthält, die die Kosten des Wurzelknotens minimieren. Unter den Konnektoren eines Knotens s ist, falls vorhanden, stets genau ein Konnektor markiert, über welchen die minimalen Kosten von s erzielt werden. $\mathcal{G}'$ wird nun berechnet, indem in $\mathcal{G}$ alle markierten Konnektoren traversiert und alle besuchten Konnektoren sowie Knoten in $\mathcal{G}'$ übernommen werden. Unter allen noch unexpandierten Knoten aus $\mathcal{G}'$ wird nun ein solcher mit einem maximalen Heuristikwert selektiert. Auf die Motivation dieser Selektionsstrategie wird in Beispiel 3.2 eingegangen. Der selektierte Knoten n wird schließlich expandiert, indem alle in ihm anwendbaren Aktionen in Konnektoren übersetzt und, zusammen mit den daraus resultierenden Knoten, in $\mathcal{G}$ übernommen werden. Jeder Knoten besitzt einen Kostenwert, der die (approximierten) Kosten des UND/ODER-Graphen mit diesem Knoten als Wurzel

darstellt. Initial werden diese Kosten durch eine Heuristik geschätzt. Gewonnene Knoten, also Knoten $n' \in G$, werden als gelöst markiert.

Der Rückwärtsschritt ist für die Propagierung der durch die Expansion gewonnenen neuen Informationen verantwortlich. Es sind genau drei Dinge, die gegebenenfalls aktualisiert werden können: Die Kosten der Knoten, ob ein Knoten als gelöst markiert wurde und die Markierungen der Konnektoren.
Sei m der Knoten, der im Vorwärtsschritt expandiert wurde. Dessen Kosten, $c(m)$, wurden zunächst lediglich durch eine Heuristik $h(m)$ abgeschätzt. Durch die Expansion von m stehen nun auch die Kosten von dessen Nachfolgern zur Verfügung. Folglich kann $c(m)$ auf Grundlage der Konnektoren von m, $C(m)$, aktualisiert werden. Analog zur Definition der Kosten eines UND/ODER-Graphen (vgl. Definition 2.9) seien einem Konnektor $c_i = (m, \{n_{1i}, \ldots, n_{ki}\}) \in C(m)$ die Kosten $\text{cost}(c_i) = 1 + \max_{j=1,\ldots,k} c(n_{ji})$ zugewiesen. Die Kosten des Knotens m sind schließlich über seinen günstigsten Konnektor definiert, das heißt $c(m) := \min_{c_i \in C(m)} \text{cost}(c_i)$. Sei ab nun c_i der Konnektor, über welchen dieses Minimum erzielt wird, das heißt $c_i := \text{argmin}_{c_i \in C(m)} \text{cost}(c_i)$. Da über c_i die minimalen Kosten von m erzielt werden, wird (nur) dieser Konnektor markiert. Falls alle Nachfolger des Konnektors c_i als gelöst markiert sind, kann auch m als gelöst markiert werden.
Falls sich die Kosten von m geändert haben oder falls m als gelöst markiert wurde, werden alle Knoten auf mögliche Aktualisierungen untersucht, die Vorgänger von m durch einen markierten Konnektor sind. Auf diese Weise werden so lange Informationen zurückpropagiert, bis keine weiteren Aktualisierungen mehr möglich sind.
Wurde der Wurzelknoten als gelöst markiert, kann durch Traversierung der markierten Konnektoren der Lösungsgraph gewonnen werden. Ist dieser noch nicht als gelöst markiert, wird erneut der Vorwärtsschritt ausgeführt.

Der Rückwärtsschritt ist die kritische Stelle, weswegen die hier erläuterte Standardvariante nicht mit Zyklen umgehen kann. Falls $\mathcal{G}$ nämlich nicht zyklenfrei ist, kann diese Aktualisierung gegebenenfalls unendlich lange andauern. Die Idee der verwendeten Adaption ist schlicht, in jedem Rückwärtsschritt jeden Knoten höchstens ein Mal zu aktualisieren.

1 Erstelle Suchgraphen $\mathcal{G}$, der lediglich aus dem Startknoten s_0 besteht. Assoziiere mit ihm die Kosten $c(s_0) = h(s_0)$, wobei h eine beliebige Heuristikfunktion ist. Falls $s_0 \in G$, kennzeichne s_0 als *gelöst*.

2 **while** s_0 *nicht als gelöst gekennzeichnet* **do**

3 Berechne einen Teillösungsgraphen $\mathcal{G}'$ von $\mathcal{G}$, indem markierte Konnektoren von $\mathcal{G}$ ausgehend von s_0 traversiert und in $\mathcal{G}'$ übernommen werden.

4 Selektiere Blattknoten $n \notin G$.

5 Expandiere Knoten n und füge alle seine Nachfolger in $\mathcal{G}$ ein. Weise jedem dieser Nachfolger n_j, der noch nicht in $\mathcal{G}$ vorhanden war, die Kosten $c(n_j) = h(n_j)$ zu. Kennzeichne jeden dieser Nachfolger n_j als *gelöst*, falls $n_j \in G$.

6 Erstelle Menge $U = \{n\}$.

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

8 Entferne aus U einen Knoten m, so dass m keine Nachfolger in $\mathcal{G}$ hat, die in U vorkommen.

9 Aktualisiere die Kosten $c(m)$ wie folgt: Für jeden Konnektor $c_i = (m, \{n_{1i}, \ldots, n_{ki}\})$ berechne $\text{cost}(c_i) := 1 + \max_{j=1,\ldots,k} c(n_{ji})$. Setze $c(m)$ auf $\min_{c_i \in C(m)} \text{cost}(c_i)$ und markiere den Konnektor, durch den dieses Minimum erzielt wurde. Lösche die alte Markierung, falls sie von der neuen verschieden ist. Falls alle Nachfolger durch diesen Konnektor als gelöst gekennzeichnet sind, kennzeichne auch m als gelöst.

10 Falls m als *gelöst* gekennzeichnet wurde oder falls die aktualisierten Kosten $c(m)$ verschieden von den vorhergehenden sind, übernehme in U diejenigen Vorgänger von m, so dass m einer ihrer Nachfolger durch einen markierten Konnektor ist.

11 **end**

12 **end**

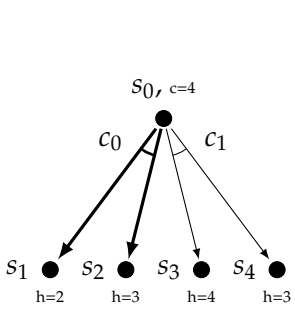
Algorithmus 3.1 : AO*-Algorithmus

3.2. Beispiel zum AO*-Algorithmus

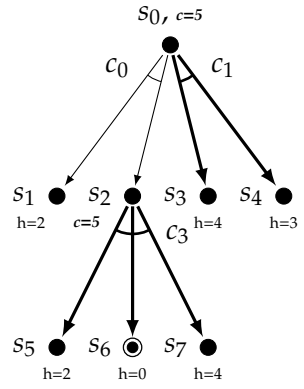
Um die Vorgehensweise des AO*-Algorithmus zu verstehen, betrachten wir ein einfaches Beispiel.

Abbildung 3.1 zeigt links einen UND/ODER-Graphen $\mathcal{G}$ nach der Expansion des Wurzelknotens s_0. In diesem Wurzelknoten sind die beiden Konnektoren $c_0 = (s_0, \{s_1, s_2\})$ und $c_1 = (s_0, \{s_3, s_4\})$ anwendbar, das heißt $C(s_0) = \{(s_0, \{s_1, s_2\}), (s_0, \{s_3, s_4\})\}$. Da der Konnektor c_0 billiger ist, das heißt $\min_{c \in C(s_0)} \text{cost}(c) = \min\{\text{cost}(c_0), \text{cost}(c_1)\} = \min\{1 + \max\{h(s_1), h(s_2)\}, 1 + \max\{h(s_3), h(s_4)\}\} = \min\{1 + \max\{2, 3\}, 1 + \max\{4, 3\}\} =$

$\min\{1+3, 1+4\} = \min\{4,5\} = 4$, und $\operatorname{argmin}_{c \in C(s_0)} \text{cost}(c) = c_0$, wird c_0 markiert (dargestellt durch Fettdruck).
Der Teillösungsgraph $\mathcal{G}'$, der sich aus $\mathcal{G}$ durch Traversierung von markierten Konnektoren ergibt, enthält genau die beiden Blätter s_1 und s_2. Unter diesen Blättern wird ein solches zur Expansion ausgewählt, das einen maximalen Heuristikwert besitzt. Der Sinn dieser Strategie wird nachfolgend diskutiert. Die Expansion des Knotens s_2 führt schließlich zum Graphen, der rechts in Abbildung 3.1 dargestellt ist. Da $c_3 = (s_2, \{s_5, s_6, s_7\})$ der einzige Konnektor des Knotens s_2 ist, wird dieser markiert. Dessen Kosten sind $\text{cost}(c_3) = \min\{1 + \max\{2, 0, 4\}\} = 5$. Die Kosten c des Knotens s_2, $c(s_2)$, die initial durch $h(s_2) = 3$ abgeschätzt wurden, können nun auf $c(s_2) = 5$ heraufgesetzt werden.
Alle Knoten, die Vorgänger von s_2 durch einen markierten Konnektor sind, werden auf notwendige Aktualisierungen überprüft. Nun gilt $\min_{c \in C(s_0)} \text{cost}(c) = 5$ mit $\operatorname{argmin}_{c \in C(s_0)} \text{cost}(c) = c_1$, das heißt die Kosten von s_0, $c(s_0)$, können auf 5 heraufgesetzt werden; außerdem wird die Markierung von c_0 auf c_1 umgesetzt.
Diese Vorgehensweise wird fortgesetzt, bis entweder ein Lösungsgraph gefunden wurde oder keine weiteren Expansionen mehr möglich sind.



UND/ODER-Graph nach Expansion von s_0.



UND/ODER-Graph nach Expansion von s_2.

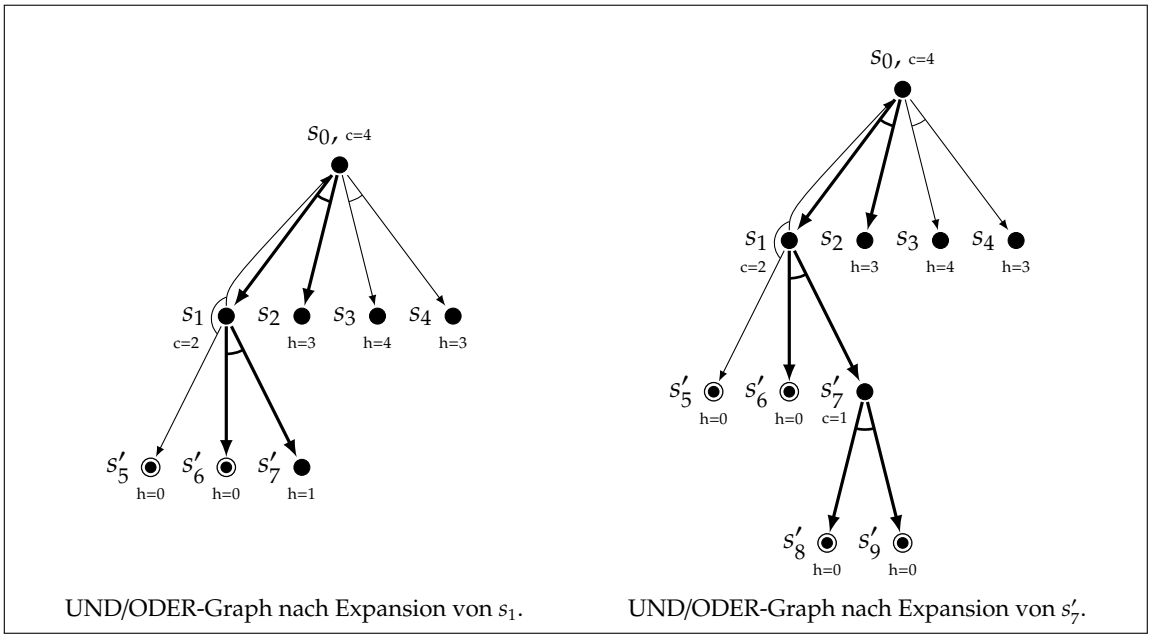
Abbildung 3.1.: Expansion des Knotens mit größtem Heuristikwert.

Nun wird der Sinn der Selektionsstrategie, stets einen Knoten zu expandieren, der einen maximalen Heuristikwert besitzt, diskutiert.

Nach jeder Expansion werden gegebenenfalls Kostenwerte heraufgesetzt, was das Umsetzen von Markierungen bewirken kann. Falls die Expansion eines Knotens s eine Umsetzung der Markierung eines Konnektors c bewirkt mit $s \in succ(c)$, so spart es unter Umständen Rechenzeit, diese Umsetzung so früh wie möglich zu bewirken. Nilsson [27] schlägt ebenfalls diese Selektionsstrategie vor und bezeichnet sie als *fail first*. Betrachten wir als Beispiel den linken Graphen aus Abbildung 3.2, welcher wie das vorhergehende Beispiel aus dem Graphen $\mathcal{G}$ hervorgeht. Während in $\mathcal{G}$ zuvor ein sich in einem Teillösungsgraphen befindlicher Knoten mit *maximaler* Heuristik

expandiert wurde, nämlich s_2, wurde nun ein sich in einem Teillösungsgraphen befindlicher Knoten mit *minimaler* Heuristik expandiert, nämlich s_1. Das erneute Expandieren eines Knotens des Teillösungsgraphen mit minimaler Heuristik führt durch Expansion des Knotens s'_7 zum rechten Graphen der Abbildung 3.2. Erst jetzt wird der Knoten s_2 expandiert, was aufgrund der Heraufsetzung der Kosten von s_2 auf 5 zu einer Umsetzung der obersten Markierung führt. Sofern über den Konnektor c_1 ein Lösungsgraph hergeleitet werden kann, waren die Expansionen der Knoten s_1 und s'_7 überflüssig.

Hier ist jedoch anzumerken, dass die Strategie, stets einen Knoten mit maximalem Heuristikwert zu expandieren, nicht immer mindestens so gut sein muss wie die Strategie, stets einen Knoten mit kleinstem Heuristikwert zu expandieren. Dies ist dadurch zu begründen, dass im Allgemeinen die Güte der Heuristikfunktion nicht bekannt ist. Betrachten wir als Beispiel den Konnektor $(s'_0, \{s'_1, s'_2\})$, wobei der Heuristikwert von s'_1 besonders klein sei und der von s'_2 besonders groß. Mit der Strategie *fail first* wird der Knoten s'_2 zuerst expandiert. Falls dessen Kostenschätzung aber sehr genau ist, während die von s'_1 hingegen weit unterschätzt, und gleichzeitig gilt, dass $c^*(s'_1) > c^*(s'_2)$, wäre es besser gewesen, zuerst s'_1 zu expandieren.



UND/ODER-Graph nach Expansion von s_1. UND/ODER-Graph nach Expansion von s'_7.

Abbildung 3.2.: Expansion des Knotens mit kleinstem Heuristikwert.

4. Pattern-Database-Heuristiken

Pattern-Database-Heuristiken sind speicherbasierte Abstraktionsheuristiken. Die grundlegende Idee besteht darin, ein Planungsproblem $\mathcal{P}$ vor der eigentlichen Suche durch Abstraktion zu vereinfachen, um danach das vereinfachte Problem durch blinde Suche vollständig und optimal zu lösen. Durch diese Suche kann für jeden erreichbaren abstrakten Zustand dessen Kostenwert berechnet werden, der schließlich in der Pattern-Database abgelegt wird. Um den Informationsgehalt dieser Heuristiken zu verbessern, ist es sinnvoll, mehr als bloß *eine* Abstraktion vorzunehmen, so dass für jeden Zustand $s \in S$ eine Menge von Abstraktionen existiert. Während der Suche wird beim Erreichen eines Zustands auf die Pattern-Database zugegriffen, um die Kosten der Abstraktionen in die Berechnung der Heuristik eingehen zu lassen.

Innerhalb der Handlungsplanung wurde bereits die Unterscheidung in klassisches Planen und in nichtdeterministisches Planen vorgestellt. Klassisches Planen zeichnet sich unter anderem dadurch aus, dass nur *ein* Agent existiert. In der Handlungsplanung können aber durchaus auch Probleme interessant sein, in welchen *mehrere* Agenten (in der Regel zwei) existieren, die gegeneinander antreten, wie dies zum Beispiel in Brettspielen üblicherweise der Fall ist. Das jüngste und prominenteste Beispiel für ein Planungsproblem mit zwei Agenten ist das Brettspiel Dame, für welches von Schaeffer u.a. [33] gezeigt werden konnte, dass es bei optimaler Spielweise beider Spieler in einem Unentschieden resultiert. Auch nichtdeterministische Probleme können als Probleme mit zwei Agenten aufgefasst werden. Der zweite Agent entspricht dort der Umwelt, also dem Nichtdeterminismus, der „entscheidet", welcher der möglichen Effekte einer Aktion eintrifft.
Pattern-Database-Heuristiken wurden im Bereich mit nur einem Agenten bereits sehr ausführlich untersucht. Einige dieser Arbeiten [8, 9, 25, 14] betrachten konkrete Domänen (die meist untersuchte Domäne ist hierbei das Schiebepuzzle [26]), während andere Arbeiten [29, 10, 11, 14, 17, 12, 18] Pattern-Database-Heuristiken domänenunabhängig betrachten.
Im Bereich mit zwei Agenten wurden Pattern-Database-Heuristiken bisher kaum untersucht. Domänenspezifisch untersuchten Samadi u.a. [32] Pattern-Database-Heuristiken für die Spiele Sternhalma (chinese checkers) und Schach. Für das domänenunabhängige Lösen von Planungsproblemen stellt diese Arbeit (nach Wissen des Autors) den ersten Versuch dar, Zwei-Agenten-Spiele mit dem Ansatz des Planens durch heuristische Suche unter Verwendung von Pattern-Database-Heuristiken zu lösen.

4.1. Grundlagen

Zunächst betrachten wir die konkrete Kodierung der nichtdeterministischen Probleme. Diese Kodierung ist stark an die STRIPS-Kodierung [15] angelehnt, in welcher klassische Planungsprobleme kodiert werden können. Ebenfalls angelehnt an die STRIPS-Kodierung werden lediglich Boolesche Zustandsvariablen verwendet. Dabei gilt die sogenannte *closed world assumption*, das heißt alle Zustandsvariablen, die nicht in einem Zustand explizit angegeben sind, werden als *falsch* angesehen, alle übrigen als *wahr*. Statt von Booleschen Variablen kann daher auch von statischen Prädikaten gesprochen werden.
Insbesondere in Verbindung mit Pattern-Database-Heuristiken ist die Verwendung mehrwertiger Variablen sinnvoll, um hierdurch die Speicherplatznutzung zu optimieren. In Abschnitt 4.4 wird diese Idee kurz präzisiert.

Definition 4.1 (Kodierung des nichtdeterministischen Planungsproblems)**.**
Ein nichtdeterministisches Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ wird wie folgt kodiert:

- Sei *Var* eine endliche Menge von Zustandsvariablen. Dann ist $S = 2^{Var}$ der Zustandsraum, die Menge der Zustände von $\mathcal{P}$.
- $s_0 \in S$ ist der Initialzustand.
- A ist eine endliche Menge von Aktionen der Form $a = \langle pre, eff \rangle$ mit der Voraussetzung $pre \subseteq Var$ und den nichtdeterministischen Effekten $eff = \{\langle add_1, del_1 \rangle, \ldots, \langle add_n, del_n \rangle\}$, mit $add_i, del_i \subseteq Var$ für alle $i \in \{1, \ldots, n\}$. Für die Voraussetzung *pre* und die Effekte *eff* von a wird auch $pre(a)$ und $eff(a)$ geschrieben. Die Menge aller Effektvariablen von a, $\bigcup_{i \in \{1,\ldots,n\}} add_i \cup del_i$, wird mit $effvar(a)$ bezeichnet.
- $\Gamma : S \to 2^A$, mit $\Gamma(s) = \{ a \in A \mid pre(a) \subseteq s \}$ für alle $s \in S \backslash G$ und $\Gamma(s) = \emptyset$ für alle $s \in G$
- $\delta : S \times A \to 2^S$, mit $\delta(s, a) = \{ (s \cup add) \backslash del \mid \langle add, del \rangle \in eff(a) \}$, falls $a \in \Gamma(s)$ und undefiniert sonst, für $s \in S$ und $a \in A$.
- $G \subseteq S$ ist die nichtleere Menge von Zielzuständen.

Definition 4.2 (Pattern, Abstraktion)**.**
Ein Pattern P_i für $i \in \mathbb{N}$ ist eine Teilmenge der Zustandsvariablen *Var*, bezüglich der die Abstraktion des Planungsproblems vorgenommen wird.
Sei das Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ gegeben. Für ein Pattern P_i ist die Abstraktion $\mathcal{P}^i = (S^i, s_0^i, A^i, \Gamma^i, \delta^i, G^i)$ von $\mathcal{P}$ gegeben durch:

- $Var^i := Var \cap P_i = P_i$ und $S^i = 2^{Var^i} = 2^{P_i}$. Für jedes $s \in S$ ist $s^i := s \cap P_i$.
- $s_0^i \in S^i$ ist der Initialzustand.
- Für jedes $a \in A$ ist $a^i := \langle pre(a) \cap P_i, \{ \langle add \cap P_i, del \cap P_i \rangle \mid \langle add, del \rangle \in eff(a) \} \rangle$.
  Dann ist $A^i := \{ a^i \mid a \in A \}$.

- $\Gamma^i : S^i \to 2^{A^i}$, mit $\Gamma^i(s^i) = \{ a^i \in A^i \mid pre(a^i) \subseteq s^i \}$ für alle $s^i \in S^i$.
- $\delta^i : S^i \times A^i \to 2^{S^i}$, mit $\delta^i(s^i, a^i) = \{ (s^i \cup add) \backslash del \mid \langle add, del \rangle \in \mathit{eff}(a^i) \}$, falls $a^i \in \Gamma^i(s^i)$ und undefiniert sonst, für $s^i \in S^i$ und $a^i \in A^i$.
- $G^i := \{ g^i \mid g \in G \}$.

Obwohl die Abstraktion $\mathcal{P}^i$ eines nichtdeterministischen Problems $\mathcal{P}$ bezüglich eines Patterns P_i intuitiv lediglich der Einschränkung des Problems auf eine kleinere Variablenmenge entspricht, unterscheidet sich die Definition eines abstrakten Planungsproblems $\mathcal{P}^i$ geringfügig von der eines nichtabstrakten Planungsproblems $\mathcal{P}$. Während in $\mathcal{P}$ in Zielzuständen keine Aktionen anwendbar sein dürfen, ist diese Einschränkung für $\mathcal{P}^i$ aufgehoben (vergleiche dazu Definition von Γ bzw. von Γ^i). Um Korrektheit zu gewährleisten, ist dies zwingend erforderlich, was in Beispiel 4.2 illustriert wird.

Lemma 4.1.
Sei $s \in S$. Gilt $a \in \Gamma(s)$, so gilt $a^i \in \Gamma^i(s^i)$.

Beweis.
Es gilt $a \in \Gamma(s)$. Also ist $pre(a) \subseteq s$, was $pre(a) \cap P_i \subseteq s \cap P_i = pre(a^i) \subseteq s^i$ impliziert. Damit gilt also $a^i \in \Gamma^i(s^i)$. □

In diesen Beweis ging der Unterschied zwischen der Definition von Γ und Γ^i ein: Für $s \in G$ wurde gefordert, dass $\Gamma(s) = \emptyset$. Wäre auch $\Gamma^i(s^i) = \emptyset$ für $s^i \in G^i$ gefordert worden, würde die Anwendbarkeit von a nicht die Anwendbarkeit von a^i implizieren.

Für einen Zustand $s \in S$ soll ein zulässiger Heuristikwert $h(s)$ ermittelt werden, also ein Heuristikwert, der die optimalen Kosten c^* nicht überschätzt. Eine Heuristik h ist folglich zulässig, falls für alle $s \in S$ gilt, dass $h(s) \leq c^*(s)$, woraus $h(s) = 0$ folgt für alle $s \in G$. Die Konstruktion der Heuristikfunktion h wird auf Grundlage geeignet gewählter Abstraktionen des Planungsproblems $\mathcal{P}$ erfolgen. Für $\mathcal{P}$ und ein gegebenes Pattern P_i wird das abstrakte Planungsproblem $\mathcal{P}^i$ optimal gelöst, das heißt für jeden im abstrakten Zustandsraum erreichbaren Zustand s^i werden dessen Kosten berechnet, die mit $h^i(s^i)$ bezeichnet werden.

Definition 4.3 (Heuristikfunktion h^i von $\mathcal{P}^i$)**.**
Für jedes Pattern P_i und für alle $s^i \in S^i$ sei $h^i(s^i) := c^*(s^i)$ von $\mathcal{P}^i$.

Korollar 4.1.
Für jedes Pattern P_i und für alle $s \in G$ gilt $s^i \in G^i$ und damit $h^i(s^i) = 0$. □

Es wird gezeigt werden (Korollar 4.8), dass die Heuristikfunktion h^i von $\mathcal{P}^i$ korrekt ist (das heißt dass jeder Zustand $s \in S$ tatsächlich verloren ist, falls ein Pattern P_i existiert, so dass s^i verloren ist, das heißt mit $h^i(s^i) = \infty$). Dennoch werden bereits an dieser Stelle Details diskutiert, die die Korrektheit anbelangen und im folgenden Satz festgehalten werden. Wieso wird durch die Abstraktion (also durch eine Vereinfachung) der Nichtdeterminismus nicht stärker? Mit anderen Worten: Wieso entstehen durch die

Vereinfachung des Planungsproblems bei einer gegebenen Aktion nicht weitere Nachfolger dieser Aktion, die unter Umständen die Korrektheit dahingehend verletzen, dass ein Zustand fälschlicherweise als verloren gekennzeichnet wird? Die Antwort auf diese Frage ist rein syntaktischer Natur: Falls eine Aktion $a \in A$ n Effekte besitzt, so kann a^i nach Konstruktion ebenfalls maximal n Effekte besitzen. Ein weiterer interessanter Diskussionspunkt bezüglich Korrektheit betrifft die Definition der Abbildung δ^i. Es sei ein Pattern P_i, ein Zustand $s \in S$, eine Aktion $a \in A$ und die Menge der Nachfolgezustände von a angewendet auf s, $\delta(s,a)$, gegeben. Es darf keine Rolle spielen, in welcher Reihenfolge die Abstraktion durchgeführt wird. Das heißt dass dasselbe Ergebnis erzielt werden muss, unabhängig davon, ob man die Abstraktionen der Zustände aus $\delta(s,a)$ bildet, oder ob man die Abstraktionen von s und a bildet und schließlich a^i auf s^i anwendet. Dass dies tatsächlich keine Rolle spielt, wird im nächsten Satz festgehalten und in Abbildung 4.1 grafisch dargestellt.

Satz 4.1.
Sei ein Pattern P_i gegeben, sowie $s \in S$ und $a \in \Gamma(s)$. Dann gilt: Falls $\delta(s,a) = \{s_1, \ldots, s_n\}$, so gilt $\delta^i(s^i, a^i) = \{s_1^i, \ldots, s_n^i\}$.

Beweis.
Sei $s \in S$, $a \in A$ und das Pattern P_i gegeben.
Sei weiterhin $\delta(s,a) = \{s_1, \ldots, s_n\} = \{\ (s \cup add)\backslash del \mid \langle add, del\rangle \in \mathit{eff}(a)\ \}$. Dann ist $s^i = s \cap P_i$ und $a^i = \langle \mathit{pre}(a) \cap P_i, \{\ \langle add \cap P_i, del \cap P_i\rangle \mid \langle add, del\rangle \in \mathit{eff}(a)\ \}$. $\delta^i(s^i, a^i)$ ist definiert, da $a^i \in \Gamma^i(s^i)$ nach Lemma 4.1 erfüllt ist. Weiterhin gilt:

$$
\begin{aligned}
\delta^i(s^i, a^i) &= \{\ (s^i \cup add)\backslash del \mid \langle add, del\rangle \in \mathit{eff}(a^i)\ \} \\
&= \{\ (s^i \cup (add \cap P_i))\backslash(del \cap P_i) \mid \langle add, del\rangle \in \mathit{eff}(a)\ \} \\
&= \{\ ((s \cap P_i) \cup (add \cap P_i))\backslash(del \cap P_i) \mid \langle add, del\rangle \in \mathit{eff}(a)\ \} \\
&= \{\ ((s \cup add)\backslash del) \cap P_i \mid \langle add, del\rangle \in \mathit{eff}(a)\ \} \\
&= \{s_1^i, \ldots, s_n^i\}
\end{aligned}
$$

□

Satz 4.1 ist äquivalent zur Aussage, dass das Diagramm aus Abbildung 4.1 kommutiert.

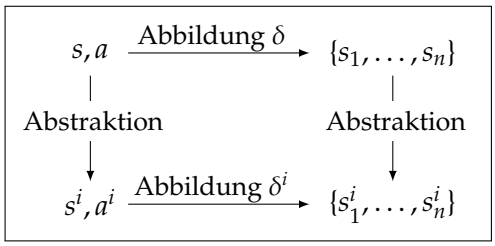


Abbildung 4.1.: Grafische Darstellung des Satzes 4.1.

Sowohl der vollständige UND/ODER-Graph von $\mathcal{P}^i$ als auch dessen Lösungsgraphen sind bereits durch die Definitionen 2.7 und 2.8 gegeben (definiert über $\mathcal{P}^i$, statt über $\mathcal{P}$). Dennoch wird die Definition des vollständigen UND/ODER-Graphen von $\mathcal{P}^i$ erneut angeführt, da sie um die Abstraktion eines Konnektors erweitert wird.

Definition 4.4 (Vollständiger UND/ODER-Graph von $\mathcal{P}^i$)**.**
Sei $\mathcal{G}' = (V', C')$ mit $V' = S^i$ und $C' = \{ (s^i, \delta^i(s^i, a^i)) \mid s^i \in S^i \text{ und } a^i \in \Gamma^i(s^i) \}$. Der vollständige UND/ODER-Graph eines nichtdeterministischen Planungsproblems $\mathcal{P}^i$ ist nun die Zusammenhangskomponente von $\mathcal{G}'$, die s^i_0 enthält.
Sei $\mathcal{P}$ das Planungsproblem, dessen Abstraktion $\mathcal{P}^i$ ist. Sei weiterhin $\mathcal{G} = (V, C)$ dessen vollständiger UND/ODER-Graph. Sei $s \in S$ und ein $a \in \Gamma(s)$ gegeben. Sei entsprechend $c \in C$ der Konnektor $(s, \delta(s, a))$. Nach Lemma 4.1 gilt damit auch $a^i \in \Gamma^i(s^i)$. Daher existiert auch der Konnektor $(s^i, \delta^i(s^i, a^i)) \in C^i$. Mit dieser Motivation sei für ein Pattern P_i und jeden Konnektor $c = (s, \delta(s, a)) \in C$ mit $s \in S$ und $a \in \Gamma(s)$ dessen Abstraktion durch $c^i := (s^i, \delta^i(s^i, a^i))$ definiert.

Das folgende Korollar stellt eine Erweiterung von Lemma 4.1 dar.

Korollar 4.2.
Sei $s_1, s_2 \in S$. Falls eine Aktion $a \in \Gamma(s_1)$ existiert mit $s_2 \in \delta(s_1, a)$, dann gilt für alle Patterns P_i, dass $a^i \in \Gamma^i(s^i_1)$ mit $s^i_2 \in \delta^i(s^i_1, a^i)$.

Beweis.
$a \in \Gamma(s_1)$ impliziert $a^i \in \Gamma^i(s^i_1)$, wie in Lemma 4.1 bereits gezeigt wurde. Da außerdem $s_2 \in \delta(s_1, a)$ gilt, ist nach Satz 4.1 auch $s^i_2 \in \delta^i(s^i_1, a^i)$ erfüllt. □

Korollar 4.3.
Sei $s_1, s_n \in S$. Falls eine Historie von s_1 nach s_n existiert, das heißt falls es eine Folge von Aktionen $a_1, \ldots, a_{n-1}$ gibt mit $a_j \in \Gamma(s_j)$ und $s_{j+1} \in \delta(s_j, a_j)$ für alle $j \in \{1, \ldots, n-1\}$, dann existiert auch die Folge von Aktionen $a^i_1, \ldots, a^i_{n-1}$ mit $a^i_j \in \Gamma(s^i_j)$ und $s^i_{j+1} \in \delta(s^i_j, a^i_j)$ für alle $j \in \{1, \ldots, n-1\}$ und alle Patterns P_i. □

Ein Zustand $s \in S$ (bzw. $s^i \in S^i$) wird *erreichbar* genannt, falls eine Historie vom Initialzustand von $\mathcal{P}$ zu s existiert (bzw. falls eine Historie vom Initialzustand von $\mathcal{P}^i$ zu s^i existiert).

Korollar 4.4.
Für jeden erreichbaren Zustand $s \in S$ ist für jedes Pattern P_i der abstrakte Zustand s^i erreichbar. Dazu ist äquivalent: Falls für ein beliebiges Pattern P_i ein Zustand $s^i \in S^i$ nicht erreichbar ist, so sind auch alle Zustände $s \in S$ mit $s \cap P_i = s^i$ nicht erreichbar. □

4.2. Beispiel zur Abstraktion mit Patterns

Es sei das nichtdeterministische Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ gegeben. Da Γ auf dem gesamten Zustandsraum definiert ist und da diese Abbildung weiterhin bereits

implizit durch Definition 4.1 gegeben ist, wird auf ihre explizite Angabe verzichtet und stattdessen auf die entsprechende Definition verwiesen. Analog ist auch δ nur implizit durch dieselbe Definition gegeben. Weiterhin sind:

- $Var = \{a, b, c, d, e\}$ und folglich $S = 2^{\{a,b,c,d,e\}}$,
- $s_0 = \{a\}$,
- $A = \{a_1, a_2\}$ mit $a_1 = \langle\{a\}, \{\langle\{b\}, \{a\}\rangle, \langle\{c\}, \{a\}\rangle\}\rangle$ und $a_2 = \langle\{b\}, \{\langle\{d\}, \emptyset\rangle, \langle\{e\}, \emptyset\rangle\}\rangle$,
- $G = \{\{b, d\}, \{b, e\}, \{c\}\}$.

Der vollständige UND/ODER-Graph von $\mathcal{P}$ ist in Abbildung 4.2 dargestellt. Dieser stellt gleichzeitig auch den einzigen und damit optimalen Lösungsgraphen von $\mathcal{P}$ mit Kosten 2 dar.
Ein Zustand $\{a_1, \ldots, a_n\}$ wird durch das Wort $a_1 \ldots a_n$ abgekürzt. ε bezeichnet das leere Wort, welches die leere Menge repräsentiert.

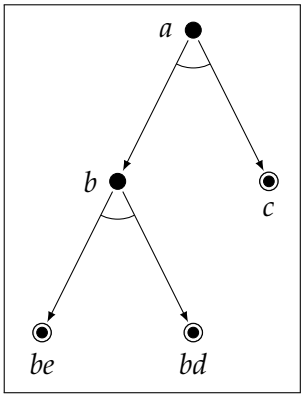


Abbildung 4.2.: Vollständiger UND/ODER-Graph von $\mathcal{P}$.

Während der Suche mit dem AO*-Algorithmus werden Heuristiken für alle während der Suche besuchten Zustände verwendet, in diesem Fall für die Zustände $\{a\}$, $\{b\}$, $\{c\}$, $\{b, e\}$ und $\{b, d\}$. Statt während der Suche Heuristikwerte für diese Zustände zu berechnen, werden sie auf Grundlage geeignet gewählter Patterns *vor* der Suche vorberechnet und in einer Pattern-Database abgelegt. Bei Bedarf werden diese zur Laufzeit in möglichst kurzer Zeit (je nach Implementierung ist der Abruf eines einzelnen Heuristikwerts $h^i(s^i)$ in $O(|s|)$ möglich für $s \in S$) aus dieser Database abgerufen.

Alle Teilmengen von S stellen mögliche Patterns dar. Wir betrachten exemplarisch drei mögliche Patterns P_1, P_2, P_3 und die damit einhergehenden Abstraktionen $\mathcal{P}^1, \mathcal{P}^2, \mathcal{P}^3$ sowie die daraus resultierenden Heuristikwerte. Wie bereits in der Definition des Planungsproblems $\mathcal{P}$, wird auch für alle Abstraktionen $\mathcal{P}^i$ und $i \in \{1, 2, 3\}$ weder Γ^i noch δ^i explizit angegeben, da sie implizit durch Definition 4.2 gegeben sind.

1. Sei $P_1 = \{b, d\}$. Dann ist $\mathcal{P}^1 = (S^1, s_0^1, A^1, \Gamma^1, \delta^1, G^1)$ mit:
   - $Var^1 = P_1 = \{b, d\}$ und $S^1 = 2^{\{b,d\}}$,
   - $s_0^1 = \emptyset$,
   - $A^1 = \{a_1^1, a_2^1\}$ mit $a_1^1 = \langle \emptyset, \{\langle \{b\}, \emptyset \rangle, \langle \emptyset, \emptyset \rangle\}\rangle$ und $a_2^1 = \langle \{b\}, \{\langle \{d\}, \emptyset \rangle, \langle \emptyset, \emptyset \rangle\}\rangle$,
   - $G^1 = \{\{b, d\}, \{b\}, \emptyset\}$.
2. Sei $P_2 = \{a, b\}$. Dann ist $\mathcal{P}^2 = (S^2, s_0^2, A^2, \Gamma^2, \delta^2, G^2)$ mit:
   - $Var^2 = P_2 = \{a, b\}$ und $S^2 = 2^{\{a,b\}}$,
   - $s_0^2 = \{a\}$,
   - $A^2 = \{a_1^2, a_2^2\}$ mit $a_1^2 = \langle \{a\}, \{\langle \{b\}, \{a\} \rangle, \langle \emptyset, \{a\} \rangle\}\rangle$ und $a_2^2 = \langle \{b\}, \{\langle \emptyset, \emptyset \rangle\}\rangle$,
   - $G^2 = \{\{b\}, \emptyset\}$.
3. Sei $P_3 = \{a, d, e\}$. Dann ist $\mathcal{P}^3 = (S^3, s_0^3, A^3, \Gamma^3, \delta^3, G^3)$ mit:
   - $Var^3 = P_3 = \{a, d, e\}$ und $S^3 = 2^{\{a,d,e\}}$,
   - $s_0^3 = \{a\}$,
   - $A^3 = \{a_1^3, a_2^3\}$ mit $a_1^3 = \langle \{a\}, \{\langle \emptyset, \{a\} \rangle\}\rangle$ und $a_2^3 = \langle \emptyset, \{\langle \{d\}, \emptyset \rangle, \langle \{e\}, \emptyset \rangle\}\rangle$,
   - $G^3 = \{\{d\}, \{e\}, \emptyset\}$.

Die vollständigen UND/ODER-Graphen dieser drei Patterns sind in Abbildung 4.3 dargestellt.

Für jeden Zustand s^i einer Abstraktion $\mathcal{P}^i$ wird dessen optimaler Kostenwert $c^*(s^i)$ von $\mathcal{P}^i$, also $h^i(s^i)$, berechnet und gespeichert. Während der Suche im nichtabstrakten Zustandsraum wird für jeden besuchten nichtabstrakten Zustand $s \in S$ die Menge seiner Abstraktionen betrachtet, in diesem Beispiel s^1, s^2 und s^3, um deren Kostenwerte/Heuristiken $h^1(s^1)$, $h^2(s^2)$ und $h^3(s^3)$ in die Berechnung der Heuristik $h(s)$ einfließen zu lassen. Auf welche Weise dies geschieht, wird in Abschnitt 4.4 erläutert.

Für die drei Abstraktionen können die folgenden Heuristikwerte berechnet werden.

s^1	$\emptyset$	$\{b\}$	$\{b, d\}$
$h^1(s^1)$	0	0	0

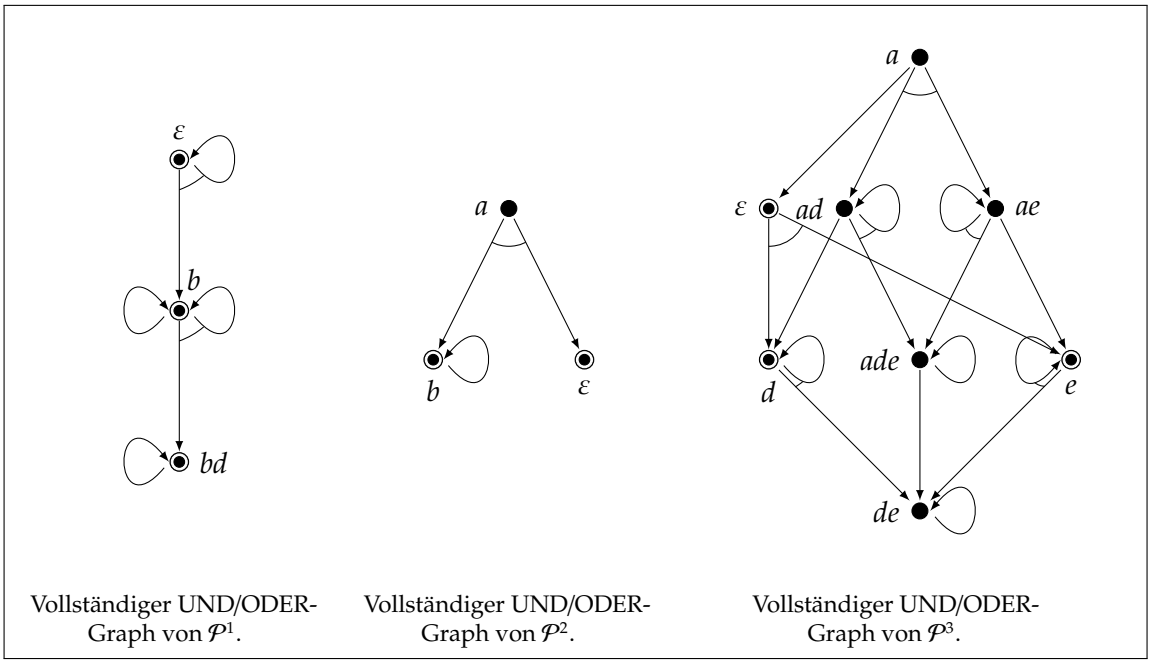
,

s^2	$\{a\}$	$\{b\}$	$\emptyset$
$h^2(s^2)$	1	0	0

,

s^3	$\{a\}$	$\emptyset$	$\{a, d\}$	$\{a, e\}$	$\{d\}$	$\{a, d, e\}$	$\{e\}$	$\{d, e\}$
$h^3(s^3)$	1	0	1	1	0	∞	0	∞

Für die Patterns $P_1 = \{b, d\}$, $P_2 = \{a, b\}$, $P_3 = \{a, d, e\}$ und die erreichbaren Nichtzielzustände $\{a\}$ und $\{b\}$ gilt $h^1(\{a\}^1) = h^1(\emptyset) = 0$, $h^1(\{b\}^1) = h^1(\{b\}) = 0$, $h^2(\{a\}^2) = h^2(\{a\}) = 1$, $h^2(\{b\}^2) = h^2(\{b\}) = 0$, $h^3(\{a\}^3) = h^3(\{a\}) = 1$ und $h^3(\{b\}^3) = h^3(\emptyset) = 0$. Für die erreichbaren Zielzustände $\{b, e\}, \{b, d\}$ und $\{c\}$ sind die Heuristikwerte für alle Heuristiken h^1, h^2 und h^3 gleich 0 (nach Lemma 4.1).



Vollständiger UND/ODER-Graph von $\mathcal{P}^1$.

Vollständiger UND/ODER-Graph von $\mathcal{P}^2$.

Vollständiger UND/ODER-Graph von $\mathcal{P}^3$.

Abbildung 4.3.: Die vollständigen UND/ODER-Graphen der Abstraktionen $\mathcal{P}^1, \mathcal{P}^2$ und $\mathcal{P}^3$.

An diesem Beispiel erkennt man, dass es zwingend notwendig wird, auch Zielzustände weiter zu expandieren. Wäre in $\mathcal{P}^1$ der gewonnene Initialzustand $\{a\}^1 = \emptyset$ nicht weiter expandiert worden, so hätte für $\{b\}^1 = \{b\}$ nicht der Heuristikwert $h^1(\{b\}) = 0$ in die Pattern-Database aufgenommen werden können. Durch die Annahme, dass alle nicht in der Pattern-Database enthaltenen Zustände verloren sind, würde der Zustand $\{b\}$ ebenfalls als verloren angesehen werden. Hierdurch wäre die Korrektheit verletzt worden, da in der nichtabstrakten Suche der Zustand $\{b\}$ als verloren gekennzeichnet worden wäre.

4.3. Additivität

Im heuristischen Planen ist man stets daran interessiert, die optimalen Kosten c^* so genau wie möglich abzuschätzen. Zusätzlich ist *Zulässigkeit* der Heuristik interessant, da je nach verwendetem Suchalgorithmus die Verwendung von zulässigen Heuristiken das Finden einer optimalen Lösung garantiert (sofern eine Lösung existiert). Im klassischen Planen ist es möglich, eine Menge von Patterns $P_1, \ldots, P_k$ so zu wählen, dass für einen Zustand $s \in S$ die resultierenden Heuristikwerte aller Abstraktionen bezüglich $P_1, \ldots, P_k$ addiert werden können, ohne dabei Zulässigkeit zu verletzen. Solche Patterns nennt man *additiv*; die entsprechenden Pattern-Database-Heuristiken heißen *additive Pattern-Database-Heuristiken*. Korf und Felner [25] (vergleiche aber auch Edelkamp [10]) nennen ein allgemeines und leicht zu überprüfendes Kriterium, das, falls erfüllt, gewährleistet,

dass die Patterns $P_1, \ldots, P_k$ additiv sind. Im Folgenden wird gezeigt werden, dass dieses Kriterium auch auf nichtdeterministische Planungsprobleme übertragbar ist.

Definition 4.5 (Additive Patterns)**.**
Die Menge $\{P_1, \ldots, P_k\}$ mit $k \in \mathbb{N}$ heißt additiv, falls für alle Zustände $s \in S$ gilt $\sum_{i=1}^{k} h^i(s^i) \leq c^*(s)$.

Definition 4.6 (Dominanz)**.**
Eine Heuristik $h_1 : S \to \mathbb{N} \cup \{0\}$ dominiert eine Heuristik $h_2 : S \to \mathbb{N} \cup \{0\}$ genau dann, wenn $h_1(s) \geq h_2(s)$ für alle $s \in S$.

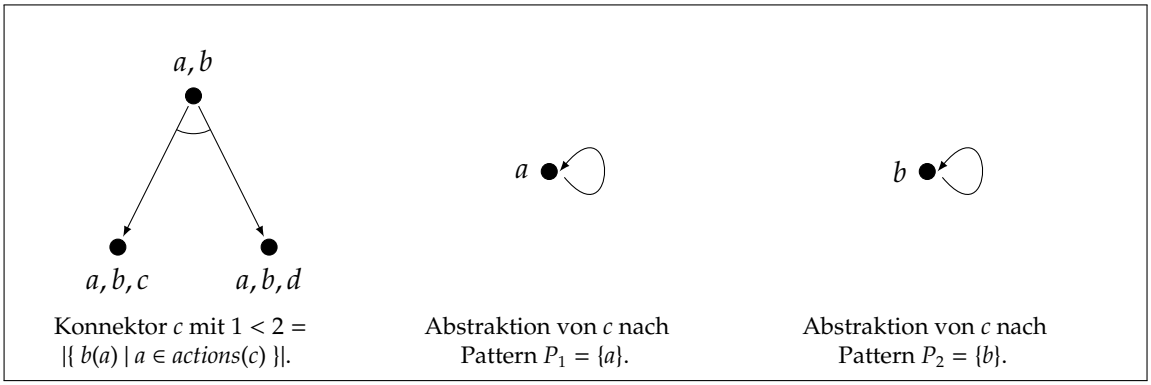
Korollar 4.5.
Falls $\{P_1, \ldots, P_k\}$ mit $k \in \mathbb{N}$ additiv ist, so ist die Heuristik $h'(s) := \sum_{i=1}^{k} h^i(s^i)$ eine zulässige Heuristik und $h'(s)$ dominiert sowohl $h^i(s^i)$ für alle $i \in \{1, \ldots, k\}$ als auch $\max_{i \in \{1,\ldots,k\}} h^i(s^i)$. □

Definition 4.7 (Aktionen eines Konnektors)**.**
Für einen UND/ODER-Graphen $\mathcal{G} = (V, C)$ mit $c \in C$ und ein Planungsproblem $\mathcal{P}$ sei durch $actions(c)$ die Menge der Aktionen bezeichnet, die dem Konnektor c entsprechen, das heißt $actions(c) := \{\, a \in \Gamma(s) \mid s = pred(c) \text{ und } \delta(s, a) = succ(c) \,\}$.
Analog sei für ein Pattern P_i, einen UND/ODER-Graphen $\mathcal{G}^i = (V^i, C^i)$ mit $c^i \in C^i$, ein Planungsproblem $\mathcal{P}$ und das Planungsproblem $\mathcal{P}^i$, $actions^i(c^i) := \{\, a^i \in \Gamma^i(s^i) \mid s^i = pred(c^i) \text{ und } \delta^i(s^i, a^i) = succ(c^i) \,\}$.

Definition 4.8 (Besitzer einer Aktion, Besitzer eines Konnektors)**.**
Für die Aktionen eines Planungsproblems $\mathcal{P}$ und die Konnektoren des vollständigen UND/ODER-Graphen $\mathcal{G} = (V, C)$ von $\mathcal{P}$ wird das Kriterium des eindeutigen Besitzers wie folgt festgelegt.
Alle Aktionen $a \in A$ haben einen eindeutigen Besitzer bezüglich der Menge $M := \{P_1, \ldots, P_k\}$, falls die Effektvariablen (vgl. Definition 4.1) jeder Aktion in maximal einem der Patterns aus M vorkommen, also falls für alle $a \in A$ und für alle $i \in \{1, \ldots, k\}$ gilt: Wenn $P_i \cap effvar(a) \neq \emptyset$, dann gilt für alle $j \in \{1, \ldots, k\}$ mit $j \neq i$, dass $P_j \cap effvar(a) = \emptyset$.
Falls die Aktionen keinen eindeutigen Besitzer bezüglich M haben, sei die Abbildung b für alle $a \in A$ und für alle $c \in C$ undefiniert. Sonst sei die Abbildung $b : A \cup C \to \{0, \ldots, k\}$ definiert, mit $b(a) = i \neq 0$ für $a \in A$, falls ein P_i existiert mit $effvar(a) \cap P_i \neq \emptyset$ und mit $b(a) = 0$, falls kein solches P_i existiert. Für $c \in C$ sei $b(c) = i$, falls für alle $a \in actions(c)$ und ein $i \in \{1, \ldots, k\}$, $b(a) = i$ gilt, und $b(c) = 0$, sonst.

Es ist erwähnenswert, dass selbst, falls alle Aktionen einen eindeutigen Besitzer haben, nicht jede Aktion einen Besitzer aus der Menge $\{1, \ldots, k\}$ haben muss, sondern aus der Menge $\{0, \ldots, k\}$. Das Kriterium des eindeutigen Besitzer sagt lediglich aus, dass die Effektvariablen einer jeden Aktion in *maximal* einem Pattern auftauchen. Prinzipiell ist es dennoch möglich, dass alle Effektvariablen einer Aktion in keinem Pattern vorkommen, obwohl alle Aktionen einen eindeutigen Besitzer haben. In einem solchen Fall ist der Besitzer der Aktion *niemand*, das heißt $b(a) = 0$, falls $effvar(a) \cap P_i = \emptyset$ für alle $i \in \{1, \ldots, k\}$.
Weiterhin erwähnenswert ist, dass nach Definition von b der Besitzer eines Konnektors

c auch dann *niemand* ist, also $b(c) = 0$, falls die Aktionen, die dem Konnektor c entsprechen, $actions(c)$, unterschiedliche Besitzer haben. Sei als Beispiel (illustriert in Abbildung 4.4) $c = (\{a, b\}, \{\{a, b, c\}, \{a, b, d\}\})$ gegeben und die beiden Aktionen a_1, a_2, die in dem Zustand $\{a, b\}$ dem Konnektor c entsprechen mit $a_1 = \langle\{a\}, \{\langle\{b, c\}, \emptyset\rangle, \langle\{b, d\}, \emptyset\rangle\}\rangle$ und $a_2 = \langle\{b\}, \{\langle\{a, c\}, \emptyset\rangle, \langle\{a, d\}, \emptyset\rangle\}\rangle$. Seien die Patterns $P_1 = \{a\}$ und $P_2 = \{b\}$ gegeben. Dann gilt $b(a_1) = 2$ und $b(a_2) = 1$, das heißt beide Aktionen haben unterschiedliche Besitzer, obwohl sie demselben Konnektor entsprechen, das heißt obwohl $actions(c) = \{a_1, a_2\}$. Wie in Lemma 4.3 gezeigt wird, fallen solche Konnektoren für alle Abstraktionen immer zu punktförmigen Zyklen zusammen, das heißt ein Konnektor $c \in C$ mit $|\{\ b(a) \mid a \in actions(c)\ \}| > 1$ wird für alle Abstraktionen c^i zu einem Zykel der Form $(s^i, \{s^i\})$ für alle $i \in \{1, \ldots, k\}$.



Konnektor c mit $1 < 2 = |\{\ b(a) \mid a \in actions(c)\ \}|$.

Abstraktion von c nach Pattern $P_1 = \{a\}$.

Abstraktion von c nach Pattern $P_2 = \{b\}$.

Abbildung 4.4.: Ein Konnektor $c = (\{a, b\}, \{\{a, b, c\}, \{a, b, d\}\})$ mit $|\{\ b(a) \mid a \in actions(c)\ \}| = 2 > 1$ und dessen Abstraktionen nach $P_1 = \{a\}$ und $P_2 = \{b\}$.

Das folgende Lemma ist offensichtlich: Falls die Abstraktion einer Aktion bezüglich eines Patterns gebildet wird, das nicht Besitzer dieser Aktion ist, so sind alle Effekte leer. Somit existiert genau ein Nachfolgezustand dieser Aktion, der mit dem identisch ist, auf den diese Aktion angewendet wurde.

Lemma 4.2.
Sei $s \in S, a \in \Gamma(s)$ und $b(a) = i$ mit $i \in \{0, \ldots, k\}$. Dann gilt für alle $j \in \{1, \ldots, k\}$ und $j \neq i$, dass $\delta^j(s^j, a^j) = \{s^j\}$.

Beweis.
Sei $s \in S, a \in \Gamma(s)$ und $b(a) = i$ mit $i \in \{0, \ldots, k\}$. Dann gilt wegen des eindeutigen Besitzers, dass für alle $j \in \{1, \ldots, k\}$ und $j \neq i$, $effvar(a) \cap P_j = \emptyset$. Per Definition gilt daher:

$$effvar(a) \cap P_j = \bigcup_{i=1}^{n} (add_i \cup del_i) \cap P_j = \emptyset \text{ mit } eff(a) = \{\langle add_1, del_1\rangle, \ldots, \langle add_n, del_n\rangle\}$$

Somit gilt:

$$a^j = \langle pre(a) \cap P_j, \{\ \langle add \cap P_j, del \cap P_j\rangle \mid \langle add, del\rangle \in eff(a)\ \}$$

$$= \langle pre(a) \cap P_j, \{\langle \emptyset, \emptyset \rangle\} \rangle$$

Damit folgt $\delta^j(s^j, a^j) = \{s^j\}$. □

Mit Hilfe des vorangegangenen Lemmas kann das folgende Lemma leicht gezeigt werden, welches besagt, dass jeder Konnektor $c \in C$ höchstens für dasjenige Pattern P_i nicht zu einem punktförmigen Zykel wird, das der eindeutige Besitzer dieses Konnektors ist.

Lemma 4.3.
Es sei $c = (s, \{s_1, \ldots, s_n\}) \in C$ ein Konnektor des vollständigen UND/ODER-Graphen (V, C) von $\mathcal{P}$ und alle $a \in A$ haben einen eindeutigen Besitzer bezüglich der Menge $\{P_1, \ldots, P_k\}$. Dann gilt für alle $j \in \{1, \ldots, k\} \backslash \{b(c)\}$, dass $c^j = (s^j, \{s^j\})$.

Beweis.
Sei der betrachtete Konnektor $c = (s, \{s_1, \ldots, s_n\}) \in C$.

Fall 1: Sei $b(c) = i \in \{0, \ldots, k\}$ und $|\{\ b(a) \mid a \in actions(c)\ \}| = 1$.
Folglich gilt für alle $a \in actions(c)$, dass $b(a) = i$. Nach Lemma 4.2 gilt für alle $j \in \{1, \ldots, k\}$ mit $j \neq i$, dass $\delta^j(s^j, a^j) = \{s^j\}$. Also ist $c^j = (s^j, \{s^j\})$.

Fall 2: Sei $|\{\ b(a) \mid a \in actions(c)\ \}| > 1$ (und damit $b(c) = 0$).
Nach Definition von $actions^j(c^j)$ gilt für alle $a_1^j, a_2^j \in actions^j(c^j)$, dass $\delta^j(s^j, a_1^j) = \delta^j(s^j, a_2^j)$. Mit anderen Worten: Für einen Konnektor $c \in C$ und ein beliebiges Pattern P_j sind alle Nachfolger von c bezüglich P_j, $succ(c^j)$, für alle Aktionen $a^j \in actions^j(c^j)$ dieselben. Da $|\{\ b(a) \mid a \in actions(c)\ \}| > 1$, gibt es zwei Aktionen $a_1, a_2 \in actions(c)$ mit $b(a_1) \neq b(a_2)$. Es sei $b(a_1) = i_1$ und $b(a_2) = i_2$ mit $i_1, i_2 \in \{0, \ldots, k\}$ und $i_1 \neq i_2$. Dann gilt nach Lemma 4.2, dass entweder $\delta^{i_1}(s^{i_1}, a_1^{i_1}) = \{s^{i_1}\}$, oder $\delta^{i_1}(s^{i_1}, a_2^{i_1}) = \{s^{i_1}\}$ (die analoge Aussage gilt auch für P_{i_2}). Da $\delta^j(s^j, a_1^j) = \delta^j(s^j, a_2^j)$ für alle Patterns P_j und alle $a_1^j, a_2^j \in actions^j(c^j)$ gilt, folgt auch $\delta^{i_1}(s^{i_1}, a_1^{i_1}) = \delta^{i_1}(s^{i_1}, a_2^{i_1}) = \{s^{i_1}\}$ für alle $i_1 \in \{1, \ldots, k\}$. Somit gilt $c^j = (s^j, \{s^j\})$. □

Zusammenfassend sieht man, dass, gegeben die Eindeutigkeit des Besitzers der Aktionen, alle Konnektoren $c = (s, \{s_1, \ldots, s_n\}) \in C$ für alle Patterns bis auf maximal eines in einen Zykel der Form $c^j = (s^j, \{s^j\})$ zusammenfallen. Das Pattern P_i mit $i \neq j$ und $i \neq 0$, für welches c^i nicht zwangsweise ein Zykel ist, ist der Besitzer des Konnektors, das heißt das Pattern P_i mit $b(c) = i \neq 0$, wobei der Besitzer eines Konnektors das Pattern ist, das Besitzer aller Aktionen ist, die sich zu diesem Konnektor ergeben.

Korollar 4.6.
Es sei $c = (s, \{s_1, \ldots, s_n\}) \in C$ ein Konnektor des vollständigen UND/ODER-Graphen (V, C) von $\mathcal{P}$ und alle $a \in A$ haben einen eindeutigen Besitzer bezüglich der Menge $\{P_1, \ldots, P_k\}$. Dann gilt für alle $j \in \{1, \ldots, k\} \backslash \{b(c)\}$, dass $h^j(s^j) = h^j(s_1^j) = \cdots = h^j(s_n^j)$.

Beweis.
Sei der betrachtete Konnektor $c = (s, \{s_1, \ldots, s_n\}) \in C$.

Nach Lemma 4.3 gilt, dass $c^j = (s^j, \{s^j\})$ für alle $j \in \{1, \ldots, k\} \backslash \{b(c)\}$. Da $c^j = (s^j, \{s_1^j, \ldots, s_n^j\})$ gilt, folgt, dass $\{s^j\} = \{s_1^j, \ldots, s_n^j\}$. Damit folgt $h^j(s^j) = h^j(s_1^j) = \cdots = h^j(s_n^j)$ für alle $j \in \{1, \ldots, k\} \backslash \{b(c)\}$. □

Lemma 4.4.
Falls $c^*(s) < \infty$, so gilt für alle Patterns P_j, dass $c^*(s^j) < \infty$.

Beweis.
Es wird gezeigt: Falls für $s \in S$ ein Lösungsgraph mit Wurzel s existiert, so existiert auch für jedes Pattern P_j ein Lösungsgraph mit Wurzel s^j. Dies wird durch Induktion über die Struktur eines Lösungsgraphen $\mathcal{G} = (V, C)$ von $\mathcal{P}$ gezeigt.
Es wird gezeigt, dass man aus einem Lösungsgraphen $\mathcal{G}$ für jedes Pattern P_j durch Abstraktion der Zustände aus V und der Konnektoren aus C einen abstrakten Lösungsgraphen $\mathcal{G}^j$ konstruieren kann. Der auf diese Weise konstruierte Lösungsgraph $\mathcal{G}^j$ muss für das Pattern P_j kein optimaler Lösungsgraph sein. Damit soll lediglich gezeigt werden, dass die Existenz eines nichtabstrakten Lösungsgraphen für alle Patterns auch die Existenz eines abstrakten Lösungsgraphen impliziert.

Induktionsanfang:
Sei $\mathcal{G} = (\{s_0\}, \emptyset)$ der betrachtete Lösungsgraph von $\mathcal{P}$. Dann gilt für jedes Pattern P_j und $\mathcal{G}^j = (\{s_0^j\}, \emptyset)$, dass $s_0^j \in G^j$. Also ist $\mathcal{G}^j$ ein Lösungsgraph und es folgt die Behauptung.

Induktionsschritt:
Unter der Annahme, die Behauptung sei für die Lösungsgraphen $\mathcal{G}_l = (V_l, C_l)$ mit Wurzel s_l für alle $l \in \{1, \ldots, n\}$ bereits gezeigt, wird gezeigt werden, dass die Behauptung auch für den Lösungsgraphen $\mathcal{G}$ gilt, wobei $\mathcal{G} = (\{s_0\} \cup \bigcup_{l=1}^n V_l, \{(s_0, \{s_1, \ldots, s_n\})\} \cup \bigcup_{l=1}^n C_l)$, und s_0 ist Wurzel von $\mathcal{G}$. Das heißt es wird gezeigt, dass für alle Patterns P_j auch ein Lösungsgraph $\mathcal{G}^j$ mit Wurzel s_0^j existiert.

Falls für ein Pattern P_j der Wurzelkonnektor $c = (s_0, \{s_1, \ldots, s_n\})$ zu einem Zykel wird, das heißt falls c^j die Form $(s_0^j, \{s_1^j, \ldots, s_n^j\})$ hat und $s_0^j \in \{s_1^j, \ldots, s_n^j\}$ gilt, so ist die Aussage bereits gezeigt, da dann ein s_l^j mit $l \in \{1, \ldots, n\}$ mit $s_0^j = s_l^j$ existiert. Da nach Induktionsvoraussetzung für s_l^j ein Lösungsgraph existiert, existiert wegen Gleichheit der beiden Zustände auch ein Lösungsgraph für s_0^j. Damit folgt direkt $c^*(s_0^j) < \infty$.
Es sei also ein Pattern P_j gegeben und $s_0^j \notin \{s_1^j, \ldots, s_n^j\}$. $\mathcal{G}^j = (\{s_0^j\} \cup \bigcup_{l=1}^n V_l^j, \{(s_0^j, \{s_1^j, \ldots, s_n^j\})\} \cup \bigcup_{l=1}^n C_l^j)$ mit $\mathcal{G}_l^j = (V_l^j, C_l^j)$ für $l \in \{1, \ldots, n\}$ ist nun ein abstrakter Lösungsgraph. □

Satz 4.2.
Die Menge $M := \{P_1, \ldots, P_k\}$ ist additiv, falls alle Aktionen $a \in A$ eines Planungsproblems $\mathcal{P}$ einen eindeutigen Besitzer bezüglich der Menge M haben.

Beweis.
Beweis durch Induktion über die Struktur eines optimalen Lösungsgraphen $\mathcal{G} = (V, C)$ von $\mathcal{P}$.
Zu zeigen ist $\sum_{j=1}^{k} h^j(s^j) \leq c^*(s)$ für alle $s \in S$, falls alle Aktionen $a \in A$ einen eindeutigen Besitzer bezüglich der Menge $\{P_1, \ldots, P_k\}$ haben.
Es kann davon ausgegangen werden, dass ein optimaler Lösungsgraph für jedes betrachtete s existiert, da anderenfalls $c^*(s) = \infty$ gälte und damit die Behauptung bereits gezeigt wäre.

Induktionsanfang:
Sei $\mathcal{G} = (\{s_0\}, \emptyset)$ ein optimaler Lösungsgraph von $\mathcal{P}$. Dann gilt für jedes Pattern P_j und $\mathcal{G}^j = (\{s_0^j\}, \emptyset)$, dass $s_0^j \in G^j$ und damit $h^j(s_0^j) = 0$. Damit folgt die Behauptung.

Induktionsschritt:
Es sei die Behauptung bereits gezeigt für $s_1, \ldots, s_n$, wobei die entsprechenden Lösungsgraphen $\mathcal{G}_1 = (V_1, C_1), \ldots, \mathcal{G}_n = (V_n, C_n)$ seien. Das heißt es gilt per Voraussetzung $\sum_{j=1}^{k} h^j(s_l^j) \leq c^*(s_l)$ für alle $l \in \{1, \ldots, n\}$, wobei s_l Wurzel von $\mathcal{G}_l$ ist. Sei nun der betrachtete Lösungsgraph $\mathcal{G} = (\{s_0\} \cup \bigcup_{l=1}^{n} V_l, \{(s_0, \{s_1, \ldots, s_n\})\} \cup \bigcup_{l=1}^{n} C_l)$, wobei s_0 Wurzel von $\mathcal{G}$ ist. Es wird gezeigt, dass $\sum_{j=1}^{k} h^j(s_0^j) \leq c^*(s_0)$ gilt.
$\mathcal{G}$ besitzt lediglich einen Konnektor am Wurzelknoten, da es sich um einen Lösungsgraphen handelt, wobei per Definition jeder Lösungsgraph für jeden Knoten s höchstens einen ausgehenden Konnektor besitzt (keinen Konnektor genau dann, wenn $s \in G$). Es darf $s_0 \notin \{s_1, \ldots, s_n\}$ vorausgesetzt werden, da der Wurzelkonnektor anderenfalls einen Zykel erzeugen würde, was per Voraussetzung ausgeschlossen ist, da $\mathcal{G}$ einen Lösungsgraphen darstellt und damit zyklenfrei ist.

Da nach Voraussetzung alle $a \in A$ einen eindeutigen Besitzer bezüglich M haben, kann das Korollar 4.6 auf den Wurzelkonnektor $c_0 = (s_0, \{s_1, \ldots, s_n\})$ angewendet werden. Damit gilt für alle $j \in \{1, \ldots, k\} \backslash \{b(c_0)\}$, dass $h^j(s_0^j) = h^j(s_1^j) = \cdots = h^j(s_n^j)$. Es wird zunächst eine Fallunterscheidung nach $b(c_0) = 0$ und $b(c_0) \neq 0$ vorgenommen.

Fall 1: $b(c_0) = 0$
Nach Korollar 4.6 gilt $h^j(s_0^j) = h^j(s_1^j) = \cdots = h^j(s_n^j)$ für alle $j \in \{1, \ldots, k\}$. Damit gilt:

$$\sum_{j=1}^{k} h^j(s_0^j) = \sum_{j \in \{1,\ldots,k\}} h^j(s_l^j) \qquad \text{für alle } l \in \{1, \ldots, n\} \qquad \text{(F1, Z1)}$$

$$< 1 + \sum_{j \in \{1,\ldots,k\}} h^j(s_l^j) \qquad \text{für alle } l \in \{1, \ldots, n\} \qquad \text{(F1, Z2)}$$

Von dieser Stelle aus wird auf Zeile (1) der Ungleichungskette gesprungen, die Fall 1 und Fall 2 wieder zusammenführt.

Fall 2: $b(c_0) \neq 0$, das heißt $b(c_0) \in \{1, \ldots, k\}$
Sei $b(c_0) = i \in \{1, \ldots, k\}$.

$$\sum_{j=1}^{k} h^j(s_0^j) = h^i(s_0^i) + \sum_{j \in \{1,\ldots,k\} \setminus \{i\}} h^j(s_l^j) \qquad \text{für alle } l \in \{1, \ldots, n\} \qquad \text{(F2, Z1)}$$

An dieser Stelle wird erneut eine Fallunterscheidung vorgenommen und unterschieden, ob die Abstraktion von c_0 bezüglich P_i, c_0^i, einem Zykel entspricht.

Fall 2.1: $b(c_0) = i \in \{1, \ldots, k\}$ und $s_0^i \in \{s_1^i, \ldots, s_n^i\}$
Der abstrakte Wurzelkonnektor c_0^i entspricht also einem Zykel, da mindestens ein Nachfolger von s_0^i dieser Zustand selbst ist. Sei dieser Nachfolger $s_{l_1}^i$, das heißt $s_0^i = s_{l_1}^i$ und es gilt $l_1 \in \{1, \ldots, n\}$. Da alle Lösungsgraphen zyklenfrei sind, kann auch der Lösungsgraph von s_0^i nicht den Konnektor c_0^i enthalten. Dies ist aber nicht notwendig, da $s_0^i = s_{l_1}^i$ gilt und damit jeder Lösungsgraph von $s_{l_1}^i$ auch ein Lösungsgraph von s_0^i ist. Formal gilt:

$$\begin{aligned}
\sum_{j=1}^{k} h^j(s_0^j) &= h^i(s_{l_1}^i) + \sum_{j \in \{1,\ldots,k\} \setminus \{i\}} h^j(s_l^j) && \text{für alle } l \in \{1, \ldots, n\} && \text{(F2, Z2)} \\
&< 1 + h^i(s_{l_1}^i) + \sum_{j \in \{1,\ldots,k\} \setminus \{i\}} h^j(s_l^j) && \text{für alle } l \in \{1, \ldots, n\} && \text{(F2, Z3)} \\
&= 1 + \sum_{j \in \{1,\ldots,k\}} h^j(s_{l_1}^j) && && \text{(F2, Z4)}
\end{aligned}$$

Fall 2.2: $b(c_0) = i \in \{1, \ldots, k\}$ und $s_0^i \notin \{s_1^i, \ldots, s_n^i\}$
Nun wird die Abschätzung $h^i(s_0^i) \leq 1 + h^i(s_{l_2}^i)$ für $l_2 := \operatorname{argmax}_{l \in \{1,\ldots,n\}} h^i(s_l^i)$ vorgenommen. Dass diese Abschätzung korrekt ist, wird im Folgenden gezeigt.
Zunächst wird gezeigt, dass $h^i(s_0^i) \leq \min_{c \in C^i(s_0^i)} \max_{s' \in succ(c)} (h^i(s') + 1)$ gilt. Zwar ist $h^i(s_0^i)$ definiert durch c^* von $\mathcal{P}^i$ mit $c^*(s_0^i) = \lim_{l \to \infty} c_l^*(s_0^i)$, wobei $\mathcal{G}^i = (V^i, C^i)$ der vollständige UND/ODER-Graph von $\mathcal{P}^i$ ist und

$$c_0^*(s^i) = 0 \quad \text{für alle } s^i \in S^i$$

$$c_{l+1}^*(s^i) = \begin{cases} 0 & \text{falls } s^i \in G^i \\ \infty & \text{falls } C^i(s^i) = \emptyset \text{ und } s^i \notin G^i \\ \min\limits_{c \in C^i(s^i)} \max\limits_{s' \in succ(c)} (c_l^*(s') + 1) & \text{sonst} \end{cases}$$

doch kann der Fall $c^*(s_0^i) = \infty$ nicht auftreten, da dann nach Lemma 4.4 auch $c^*(s_0) = \infty$ gelten müsste, was ein Widerspruch zur Voraussetzung wäre. Der Fall $h^i(s_0^i) = 0$ kann nicht ausgeschlossen werden, aber er kann ignoriert werden, wodurch nach oben abgeschätzt wird. Folglich gilt $h^i(s_0^i) \leq \min_{c \in C^i(s_0^i)} \max_{s' \in succ(c)} (h^i(s') + 1)$. Nun wird weiter

abgeschätzt, um schließlich $h^i(s_0^i) \leq 1 + h^i(s_{l_2}^i)$ für $l_2 \coloneqq \operatorname{argmax}_{l \in \{1,\dots,n\}} h^i(s_l^i)$ zu zeigen.

$$h^i(s_0^i) \leq \min_{c \in C^i(s_0^i)} \max_{s' \in succ(c)} (h^i(s') + 1) \tag{a}$$

Sei c_0^i der abstrakte Wurzelkonnektor $(s_0^i, \{s_1^i, \dots, s_n^i\})$

$$\leq \max_{s' \in succ(c_0^i)} (h^i(s') + 1) \tag{b}$$

$$= 1 + \max_{s' \in succ(c_0^i)} h^i(s') \tag{c}$$

$$= 1 + \max_{s' \in \{s_1^i,\dots,s_n^i\}} h^i(s') \tag{d}$$

$$= 1 + h^i(s_{l_2}^i) \tag{e}$$

An dieser Stelle des Beweises ist es erwähnenswert, dass der abstrakte optimale Lösungsgraph von $\mathcal{P}^i$ nicht zwangsweise ein Teilgraph der Abstraktion des Lösungsgraphen von $\mathcal{P}$ ist. Von Zeile (a) nach (b) wird eine Abschätzung nach oben vorgenommen, indem die Abstraktion des Wurzelkonnektors c_0 gewählt wird. Den optimalen Lösungsgraphen erhält man hingegen durch Selektion des Konnektors, der den entsprechenden Ausdruck minimiert. Insbesondere kann dieses Minimum durch einen Konnektor erzielt werden, der eine Abstraktion eines Konnektors darstellt, der nicht im Lösungsgraphen von $\mathcal{P}$ enthalten ist.

Die Gleichung (F2, Z1) kann nun wie folgt erweitert werden:

$$\sum_{j=1}^{k} h^j(s_0^j) \leq 1 + h^i(s_{l_2}^i) + \sum_{j \in \{1,\dots,k\} \setminus \{i\}} h^j(s_l^j) \qquad \text{für alle } l \in \{1,\dots,n\} \tag{F2, Z2'}$$

$$= 1 + \sum_{j \in \{1,\dots,k\}} h^j(s_{l_2}^j) \tag{F2, Z3'}$$

Nun wird Fall 1 und Fall 2 zusammengeführt und zeitgleich die Induktionsvoraussetzung angewendet. Die Induktionsvoraussetzung lautet $\sum_{j=1}^{k} h^j(s_l^j) \leq c^*(s_l)$ für alle $l \in \{1,\dots,n\}$.

Die Ungleichungen (F1, Z2), (F2, Z4) und (F2, Z3′) werden zusammengeführt, indem jeweils die Induktionsvoraussetzung angewendet wird; im Fall der Gleichung (F1, Z2) auf ein beliebiges $l \in \{1,\dots,n\}$, im Fall der Gleichung (F2, Z4) mit l_1 und $s_0^i = s_{l_1^i} \in \{s_1^i,\dots,s_n^i\}$ und im Fall von (F2, Z3′) schließlich mit $l_2 = \operatorname{argmax}_{l \in \{1,\dots,n\}} h^i(s_l^i)$. Im Folgenden bezeichne $l_3 \in \{1,\dots,n\}$ stellvertretend alle drei Fälle.

$$\sum_{j=1}^{k} h^j(s_0^j) = \cdots \leq 1 + \sum_{j\in\{1,\ldots,k\}} h^j(s_{l_3}^j) \qquad \text{(F1, Z2), (F2, Z4),(F2, Z3')}$$

$$\overset{\text{IV}}{\leq} 1 + c^*(s_{l_3}) \tag{1}$$

$$\leq 1 + \max_{l\in\{1,\ldots,n\}} c^*(s_l) \tag{2}$$

$$= c^*(s_0) \tag{3}$$

Die Gleichung (1) gilt wegen Anwendung der Induktionsvoraussetzung. Der Schritt von (1) nach (2) entspricht der Abschätzung gegen das Maximum. Schließlich entspricht der Schritt von (2) nach (3) der Anwendung der Definition von c^*. Da $c_0 = (s_0, \{s_1, \ldots, s_n\})$ und $s_0 \notin \{s_1, \ldots, s_n\}$ gilt, ist $c^*(s_0) = 1 + \max\{c^*(s_1), \ldots, c^*(s_n)\}$.

Damit ist insgesamt gezeigt, dass eine Menge $M := \{P_1, \ldots, P_k\}$ von Patterns additiv ist, falls alle Aktionen $a \in A$ eines Planungsproblems $\mathcal{P}$ einen eindeutigen Besitzer bezüglich der Menge M haben. □

Korollar 4.7.
Die Heuristik h^i von $\mathcal{P}^i$ ist zulässig.

Beweis.
Satz 4.2 wird auf eine einelementige Menge angewendet. Sei also $M = \{P_i\}$ und P_i ein beliebiges Pattern. Alle Aktionen $a \in A$ haben einen eindeutigen Besitzer, da $|M| = 1$. Damit ist M additiv und es gilt $h^i(s^i) \leq c^*(s)$ für alle $s \in S$. □

Korollar 4.8.
Falls ein Pattern P_i existiert mit $s^i \in S^i$ und s^i ist verloren, das heißt $h^i(s^i) = \infty$, dann sind auch alle $s \in S$ verloren mit $s \cap P_i = s^i$.

Beweis.
Da jedes h^i zulässig ist, gilt für alle $s \in S$, dass $h^i(s^i) \leq c^*(s)$. Falls also $h^i(s^i) = \infty$ gilt, muss auch $c^*(s) = \infty$ für alle s mit $s \cap P_i = s^i$ gelten. □

Satz 4.2 (Aussage über Additivitätskriterium), zusammen mit Korollar 4.5 (Aussage über Dominanz und Zulässigkeit) besagen, unter welcher Bedingung man Patterns $P_1, \ldots, P_k$ finden kann, mit deren Hilfe man eine zulässige Heuristik konstruieren kann, die mindestens so aussagekräftig ist, wie die Heuristiken, aus denen sie zusammengesetzt ist.

4.3.1. Beispiel für additive Patterns

Betrachten wir als Beispiel für additive Patterns das folgende nichtdeterministische Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ mit:

- $S = 2^{\{a,b,c,d,e\}}$,
- $s_0 = \{a\}$,
- $A = \{a_1, \ldots, a_9\}$ mit

$$
\begin{aligned}
a_1 &= \langle \{a\}, \{\langle \{b\}, \{a\}\rangle, \langle \{c\}, \{a\}\rangle\}\rangle & a_6 &= \langle \{b,e\}, \{\langle \{c\}, \emptyset\rangle\}\rangle \\
a_2 &= \langle \{b\}, \{\langle \{e\}, \emptyset\rangle, \langle \{d\}, \emptyset\rangle\}\rangle & a_7 &= \langle \{c,e\}, \{\langle \{b\}, \emptyset\rangle\}\rangle \\
a_3 &= \langle \{c\}, \{\langle \{e\}, \emptyset\rangle, \langle \{d\}, \emptyset\rangle\}\rangle & a_8 &= \langle \{b,c,d\}, \{\langle \{e\}, \emptyset\rangle\}\rangle \\
a_4 &= \langle \{b,d\}, \{\langle \{c\}, \emptyset\rangle\}\rangle & a_9 &= \langle \{b,c,e\}, \{\langle \{d\}, \emptyset\rangle\}\rangle \\
a_5 &= \langle \{c,d\}, \{\langle \{b\}, \emptyset\rangle\}\rangle
\end{aligned}
$$

- Γ und δ wie in Definition 4.1,
- $G = \{\{b,c,d,e\}\}$

Ein Lösungsgraph von $\mathcal{P}$ ist in Abbildung 4.5 dargestellt. Die Tatsache, dass die Aktionen $a_4, \ldots, a_9$ deterministisch sind, stellt keine Einschränkung dar, dies soll lediglich das Beispiel vereinfachen.

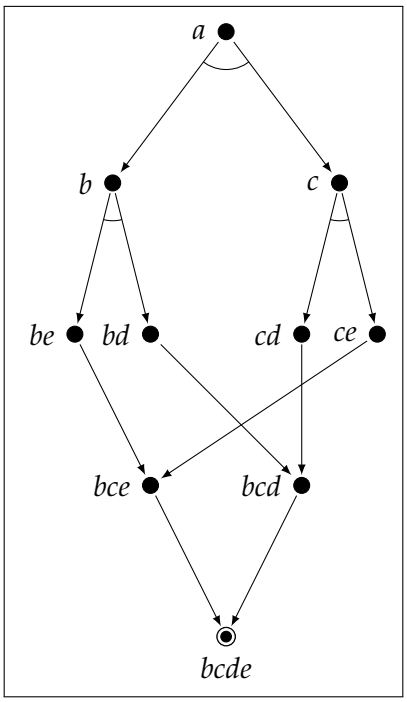


Abbildung 4.5.: Lösungsgraph von $\mathcal{P}$.

Seien nun die Patterns $P_1 = \{a, b, c\}$ und $P_2 = \{d, e\}$ gegeben. Es ergeben sich die folgenden Abstraktionen $\mathcal{P}^1$ und $\mathcal{P}^2$:

- $S^1 = 2^{\{a,b,c\}}$,
- $s_0^1 = \{a\}$,
- $A^1 = \{a_1^1, \ldots, a_9^1\}$ mit

$$
\begin{aligned}
a_1^1 &= \langle \{a\}, \{\langle \{b\}, \{a\}\rangle, \langle \{c\}, \{a\}\rangle\}\rangle \\
a_2^1 &= \langle \{b\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_3^1 &= \langle \{c\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_4^1 &= \langle \{b\}, \{\langle \{c\}, \emptyset\rangle\}\rangle \\
a_5^1 &= \langle \{c\}, \{\langle \{b\}, \emptyset\rangle\}\rangle \\
a_6^1 &= \langle \{b\}, \{\langle \{c\}, \emptyset\rangle\}\rangle \\
a_7^1 &= \langle \{c\}, \{\langle \{b\}, \emptyset\rangle\}\rangle \\
a_8^1 &= \langle \{b, c\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_9^1 &= \langle \{b, c\}, \{\langle \emptyset, \emptyset\rangle\}\rangle
\end{aligned}
$$

- Γ^1, δ^1 wie in Definition 4.2 und
- $G^1 = \{\{b, c\}\}$

- $S^2 = 2^{\{d,e\}}$,
- $s_0^2 = \emptyset$,
- $A^2 = \{a_1^2, \ldots, a_9^2\}$ mit

$$
\begin{aligned}
a_1^2 &= \langle \emptyset, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_2^2 &= \langle \emptyset, \{\langle \{e\}, \emptyset\rangle, \langle \{d\}, \emptyset\rangle\}\rangle \\
a_3^2 &= \langle \emptyset, \{\langle \{e\}, \emptyset\rangle, \langle \{d\}, \emptyset\rangle\}\rangle \\
a_4^2 &= \langle \{d\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_5^2 &= \langle \{d\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_6^2 &= \langle \{e\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_7^2 &= \langle \{e\}, \{\langle \emptyset, \emptyset\rangle\}\rangle \\
a_8^2 &= \langle \{d\}, \{\langle \{e\}, \emptyset\rangle\}\rangle \\
a_9^2 &= \langle \{e\}, \{\langle \{d\}, \emptyset\rangle\}\rangle
\end{aligned}
$$

- Γ^2, δ^2 wie in Definition 4.2 und
- $G^2 = \{\{d, e\}\}$

An dieser Gegenüberstellung ist erkennbar, dass alle Aktionen einen eindeutigen Besitzer haben, da die Effekte einer Aktion a_j^i leer sind, falls die Effekte $a_j^{\bar{i}}$ nicht leer sind mit $j \in \{1, \ldots, 9\}$ und $\bar{i} := 3 - i$ für $i \in \{1, 2\}$. Es gilt $b(a) = 1$ für alle $a \in \{a_1, a_4, a_5, a_6, a_7\}$ und $b(a) = 2$ für alle $a \in A \backslash \{a_1, a_4, a_5, a_6, a_7\}$. Die resultierenden Lösungsgraphen sind in Abbildung 4.6 dargestellt.

Damit gilt $\sum_{i=1,2} h^i(s_0^i) = \sum_{i=1,2} h^i(\{a\}^i) = h^1(\{a\}^1) + h^2(\{a\}^2) = h^1(\{a\}) + h^2(\emptyset) = 2 + 2 = 4 = c^*(\{a\})$.

Für additive Patterns kann daher durch Addition von Heuristikwerten eine informativere Heuristik gewonnen werden, ohne die Zulässigkeit zu verletzen.

4.3.2. Additivität in der Praxis

In den vorhergehenden Kapiteln wurde die Menge der Zielzustände G als Teilmenge aller Zustände definiert, das heißt $G \subseteq S$. Dies bedeutet, dass G tatsächlich *alle* Zustände enthalten muss, die Zielzustände darstellen. In der Praxis ist es fast ausschließlich so, dass für einen beliebigen Zielzustand auch alle Obermengen dieses Zustands Zielzustände sind, das heißt für jedes $g \in G$ ist auch $g' \in G$ mit $g' \supseteq g$. Dies geht darauf zurück, dass g das *eigentliche Ziel* repräsentiert, während alle anderen Variablen, die

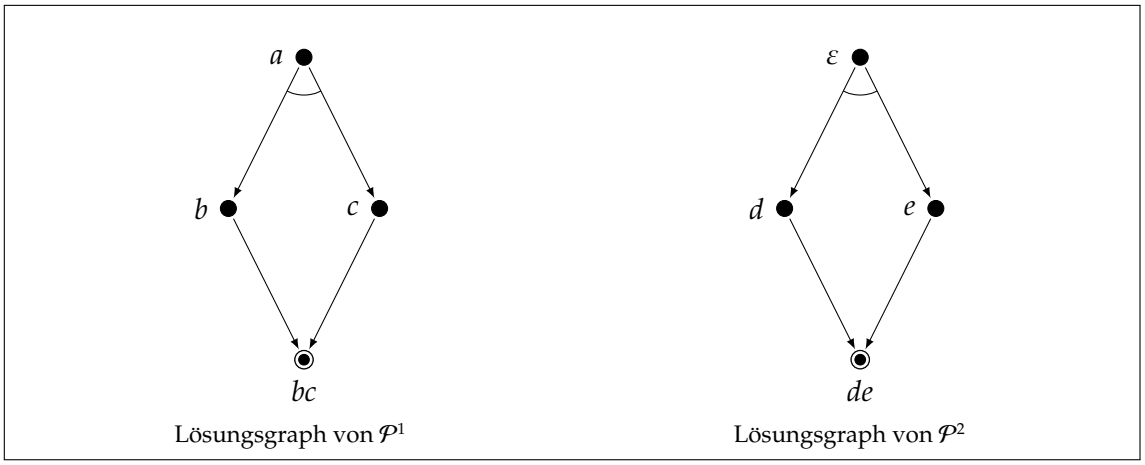


Abbildung 4.6.: Abstrakte Lösungsgraphen für die Patterns P_1 und P_2.

zusätzlich wahr oder falsch sein könnten, lediglich zusätzliche Details des aktuellen Zustands darstellen, die für das Ziel selbst aber irrelevant sind. Folglich werden in der Praxis nur wenige Zielzustände explizit repräsentiert, während alle weiteren implizit über deren Obermengen gegeben sind. Sei daher G' die Menge der explizit repräsentierten Zielzustände. Es gilt $G = \{ g' \mid g' \supseteq g \text{ für ein } g \in G' \}$. Dies unterscheidet sich von der bisherigen Definition dahingehend, dass es bislang auch möglich war, dass zwei Zustände s und g existieren mit $g \in G$, $s \notin G$ und $s \supset g$; dies ist nun nicht mehr möglich. Dies nimmt Einfluss auf die Additivität, falls $|G'| > 1$ oder falls Zielzustände existieren, die aus sehr wenigen Zustandsvariablen bestehen.

Sei $|G'| > 1$ und für alle $g_1, g_2 \in G'$ mit $g_1 \neq g_2$ gälte weder $g_1 \supseteq g_2$ noch $g_1 \subseteq g_2$. Um eine Menge M von *sinnvollen* additiven Patterns zu erhalten mit $|M| > 1$, muss jedes Pattern $P_i \in M$ mindestens eine Variable pro Zielzustand $g \in G'$ enthalten. Wäre dies nicht der Fall, das heißt gäbe es ein Pattern $P_i \in M$ und einen Zustand $g \in G'$ mit $g \cap P_i = \emptyset$, so wäre $\emptyset \in G^i$ und damit $G^i = S^i$. Damit besäße P_i keinerlei Aussagekraft, da $h^i(s^i) = 0$ für alle $s^i \in S^i$.

Betrachten wir als nächstes, wieso Zielzustände mit sehr wenigen Zustandsvariablen ein Problem für die Additivität darstellen. Sei als Beispiel ein Planungsproblem $\mathcal{P} = (S, s_0, A, \Gamma, \delta, G)$ gegeben mit $G' = \{\{g\}\}$, das heißt $\{g\} \subseteq Var$ ist der einzige explizit angegebene Zielzustand. Seien weiter die Patterns $P_1, \ldots, P_k$ gegeben.
Angenommen, g ist in keinem Pattern vorhanden, dann ist für alle Abstraktionen $\mathcal{P}^i$ mit $i \in \{1, \ldots, k\}$ die Menge der Zielzustände $G^i = \{ g' \mid g' \supseteq \emptyset \} = S^i$. Damit gilt $h^i(s^i) = 0$ für alle $s^i \in S^i$ und alle $i \in \{1, \ldots, k\}$. Damit sind die Patterns P_1 bis P_k zwar automatisch additiv, doch ist die Heuristik $\sum_{i \in \{1,\ldots,k\}} h^i(s^i)$ konstant 0.
Angenommen, g ist in genau einem Pattern P_i vorhanden. Analog zum vorherigen Fall ist dann für alle $j \neq i$ und alle $s^j \in S^j, h^j(s^j) = 0$.
Angenommen, g sei nun in mindestens zwei Patterns vorhanden. Dann ergeben sich

erneut zwei Fälle: Entweder ist g in keinem Effekt enthalten, das heißt $g \notin \mathit{effvar}(a)$, dann ist aber auch der Zielzustand nicht erreichbar oder bereits im Initialzustand erreicht, das heißt $s_0 \in G$, oder es existiert mindestens eine Aktion $a \in A$ mit $g \in \mathit{effvar}(a)$. Da g in mindestens zwei Patterns vorhanden ist, kann nicht angenommen werden, dass $\{P_1, \ldots, P_k\}$ additiv ist (da aufgrund des Fehlens des eindeutigen Besitzers Satz 4.2 nicht mehr anwendbar ist).
Zusammenfassend sieht man, dass unter der Annahme, dass für jeden Zielzustand auch alle Obermengen dieses Zustands Zielzustände darstellen, für einen einelementigen Zielzustand auf eine Menge von Patterns P_1 bis P_k entweder Satz 4.2 nicht anwendbar ist, der Additivität garantieren würde, oder $\sum_{i\in\{1,\ldots,k\}} h^i(s^i) = \max_{i\in\{1,\ldots,k\}} h^i(s^i)$ gilt, was nicht wünschenswert ist.

Es konnte also gezeigt werden, dass in der Praxis einelementige Zielzustände ungeeignet sind, um mit Hilfe von Satz 4.2 eine aussagekräftige Heuristik zu konstruieren. Diese Erkenntnis lässt sich auf mehrelementige Zielzustände verallgemeinern: Je weniger Zustandsvariablen die Zielzustände beinhalten, desto weniger Patterns können zur Addition genutzt werden. Das liegt an der Tatsache, dass für jedes Pattern mindestens eine Zustandsvariable aufgenommen werden sollte, da für jedes Pattern P_i, das keine Zielzustandsvariablen enthält, $G^i = S^i$ gilt und damit auch $h^i(s^i) = 0$ für alle $s^i \in S^i$ und für alle $i \in \{1, \ldots, k\}$.

Eine weitere Frage, die sich letztlich stellt, ist, ob diese Problematik für die Praxis tatsächlich relevant ist. Man kann schließlich $\sum_{i\in\{1,\ldots,k\}} h^i(s^i) = \max_{i\in\{1,\ldots,k\}} h^i(s^i)$ umgehen, indem die einzige Zielzustandsvariable $g \in \mathit{Var}$ in alle Patterns aufgenommen wird. Dies verletzt zwar die Voraussetzung um Satz 4.2 anwenden zu können, welcher ein Verfahren nahe legt, um eine nichtüberschätzende Heuristik zu konstruieren. Doch sollte angemerkt werden, dass zulässige Heuristiken vor allem dann bevorzugt werden, wenn der verwendete Suchalgorithmus unter Verwendung einer zulässigen Heuristik das Finden einer optimalen Lösung garantiert. Begnügt man sich mit dem Finden suboptimaler Lösungen, zeigen überschätzende Heuristiken oft deutlich bessere Ergebnisse [31].

4.4. Berechnung und Speicherung der Pattern-Database-Heuristiken

Im Folgenden wird gezeigt werden, wie für ein Planungsproblem $\mathcal{P}$ und ein Pattern P_i die Heuristikwerte berechnet und schließlich gespeichert werden.
Für $\mathcal{P}^i = (S^i, s_0^i, A^i, \Gamma^i, \delta^i, G^i)$ wird zunächst der vollständige UND/ODER-Graph berechnet. Für jeden abstrakten Zustand s^i dieses Graphen soll anschließend der optimale Kostenwert $h^i(s^i)$ bestimmt werden. Dies wird gewährleistet, indem die optimalen Kosten $h^i(s^i) = c^*(s^i) = \lim_{l\to\infty} c_l^*(s^i)$ von $\mathcal{P}^i$ durch Value-Iteration bestimmt werden. Die Berechnungsvorschrift entspricht der optimalen Kostenfunktion:

$$h_l^i(s^i) = 0 \quad \text{für alle } s^i \in S^i$$

$$h_{l+1}^i(s^i) = \begin{cases} 0 & \text{falls } s^i \in G^i \\ \infty & \text{falls } C(s^i) = \emptyset \text{ und } s^i \notin G^i \\ \min\limits_{c \in C(s^i)} \max\limits_{s' \in succ(c)} (h_l^i(s') + 1) & \text{sonst} \end{cases}$$

Da aus Korrektheitsgründen (vgl. Beispiel 4.2, letzter Absatz) der vollständige UND/ODER-Graph betrachtet wird, können auch verlorene Zustände (Kosten sind unendlich) im betrachteten abstrakten UND/ODER-Graphen enthalten sein. Wie gewährleistet werden kann, dass der Value-Iteration-Algorithmus dennoch terminiert, wird in Abschnitt 5.2 erläutert.

Sei nun für eine feste Menge von Patterns $P = \{P_1, \ldots, P_p\}$ eine Partitionierung in additive Patterns gegeben. Seien also $M_1, \ldots, M_m$ Mengen von Patterns mit $\bigcup_{i \in \{1,\ldots,m\}} M_i = P$, so dass für alle $i \in \{1, \ldots, m\}$ M_i additiv ist. Für $s \in S$ wird nun die Heuristikfunktion h über das Maximum der Heuristikwerte der Mengen M_i definiert. Hierzu werden für ein Pattern P_i abstrakte Benennungen der Form s^{P_i} und h^{P_i} mit s^i und h^i identifiziert.

Definition 4.9 (Heuristikfunktion h von $\mathcal{P}$)**.**
Für eine Menge von Patterns $P = \{P_1, \ldots, P_p\}$ und eine Partitionierung in m additive Patterns, $\bigcup_{i \in \{1,\ldots,m\}} M_i = P$, sei die Heuristikfunktion h wie folgt definiert: $h(s) := \max_{i \in \{1,\ldots,m\}} \sum_{P \in M_i} h^P(s^P)$.

Satz 4.3.
Die Heuristik h von $\mathcal{P}$ ist zulässig.

Beweis.
Nach Satz 4.2 gilt, dass $\sum_{P \in M} h^P(s^P) \leq c^*(s)$ für alle $s \in S$, falls M eine additive Menge von Patterns ist. Da nach Voraussetzung alle $M_1, \ldots, M_m$ additiv sind, folgt die Behauptung. □

Definition 4.10 (Maximal additiv)**.**
Eine Menge $M_i \subseteq P$ von Patterns heißt maximal additiv, falls kein weiteres Pattern $P_j \in P, P_j \notin M_i$ existiert, so dass $M_i \cup \{P_j\}$ additiv ist.

Sei eine Partitionierung der Patterns in die Mengen $M_1, \ldots, M_m$ gegeben. Werden die Mengen $M_1, \ldots, M_m$ so erweitert, dass die daraus resultierenden Mengen $M_1', \ldots, M_m'$ maximal additiv sind, so ergeben sich Heuristikwerte, die mindestens so groß sind, wie bei der Partitionierung in die Mengen $M_1, \ldots, M_m$, ohne jedoch Zulässigkeit zu verletzen. Hierbei sei angemerkt, dass für eine Menge M_i von Patterns mehrere maximal additive Obermengen existieren, für welche unterschiedlich gute Heuristikwerte möglich sind. Dies ist offensichtlich, wenn man $M_i = \emptyset$ und die beiden Obermengen $M_i' = \{P_1\}$, sowie $M_i'' = \{P_2\}$ mit $P_1 \neq P_2$ betrachtet.

Es wurde bereits gezeigt, dass für ein einzelnes Pattern P_i die optimalen Kosten aller erreichbaren Zustände mittels Value-Iteration bestimmt werden können. Auch ist durch die Definition von h bereits klar, wie die einzelnen Heuristikwerte (zum Beispiel $h^1(s^1)$) in die Berechnung von h eingehen. Es muss jedoch noch diskutiert werden, auf welche Art die einzelnen Heuristikwerte gespeichert werden, damit die Kriterien der Optimierung des Speicherplatzes und der Maximierung der Zugriffsgeschwindigkeit gewahrt werden.

Wir wissen bereits (Korollar 4.8), dass ein Zustand s verloren ist, falls für ein beliebiges Pattern P_i der Zustand s^i verloren ist. Da wir nach Korollar 4.4 zusätzlich wissen, dass für jeden erreichbaren Zustand $s \in S$ und jedes Pattern P_i der Zustand s^i erreichbar ist, kann die Konvention verwendet werden, dass alle abstrakten Zustände, für welche keine Heuristik explizit gespeichert wurde, verloren sind. Diese Konvention kann genutzt werden, um Speicherplatz zu sparen, indem lediglich diejenigen abstrakten Zustände gespeichert werden, die gewonnen sind, das heißt deren Heuristikwert endlich ist. Bei einem Zugriff auf einen Zustand, der nicht in der Pattern-Database vorhanden ist, wird dann automatisch angenommen, dass dieser verloren ist.

Es bleibt weiterhin die Frage zu klären, auf welche Weise die Heuristikwerte gespeichert werden. Für jeden Zustand s der Suche muss in möglichst kurzer Zeit der Heuristikwert $h^i(s^i)$ für alle verwendeten Patterns P_i abgerufen werden können. s^i kann stets schnell berechnet werden, da $s^i := s \cap P_i$. Weiterhin muss jedes s^i auf den entsprechenden Heuristikwert $h^i(s^i)$ abbilden. Je nach Wahl dieser Abbildungsvorschrift variieren Zugriffsgeschwindigkeit und Speicherplatznutzung. Gesucht ist also eine Abbildungsvorschrift $f^i : S^i \to \mathbb{N} \cup \{0\}$ mit $f^i(s^i) = h^i(s^i)$, die möglichst schnell berechnet werden kann und möglichst wenig Speicherplatz erfordert.

Eine Möglichkeit, dies zu realisieren, ist, die Abbildung f^i als Hashtabelle zu implementieren, welche wiederum eine Hashfunktion verwendet. Die Zugriffsgeschwindigkeit ist maximal, falls die verwendete Hashfunktion perfekt (also injektiv) ist. Die intuitivste Möglichkeit, eine Hashtabelle mit perfekter Hashfunktion zu erstellen, ist, ein Array A zu verwenden, in welchem der Index id des Arrays A dem Urbild der Hashtabelle (bzw. dem Bild der Hashfunktion) entspricht und der Inhalt $A[id]$ dem Bild der Hashtabelle. Um ein Array als Datenstruktur nutzen zu können, muss folglich der Schlüssel einem Zahlenwert entsprechen. Dadurch wird es notwendig, jeden abstrakten Zustand eindeutig durch eine Zahl zu repräsentieren. Dies ist durch eine einfache Auflistung aller Zustände von S^i zu erreichen, indem man die Zustandsvariablen aus Var^i beginnend bei 0 durchnummeriert, wodurch jedem $v \in V^i$ eine eindeutige Zahl $z^i(v) \in \mathbb{N} \cup \{0\}$ zugeordnet wird. Schließlich wird für einen Zustand s^i der Index durch $\sum_{v \in s^i} 2^{z^i(v)}$ berechnet. Während diese Variante eine Zugriffsgeschwindigkeit von $O(1)$ gewährleistet (ohne dem Aufwand für die Schnittmengenberechnung und die Berechnung des Indexwertes), ist der Speicherplatzbedarf exponentiell in der Anzahl der Zustandsvariablen des jeweiligen Patterns $O(2^{|P_i|})$.

Dieser Speicherbedarf kann durch Verwendung von mehrwertigen Variablen gegebenenfalls drastisch reduziert werden. Betrachten wir als Beispiel die Booleschen Variablen *box1-at-pos1*,. . . ,*box1-at-posn*, die ausdrücken, an welcher Position sich das Objekt *box1* befindet. Falls sich dieses Objekt stets an genau einer Position (*pos1* bis *posn*) befindet, müssten in die Pattern-Database alle n Zustandsvariablen aufgenommen werden, um alle möglichen Positionen zu kodieren. Unter Verwendung von mehrwertigen Variablen wäre für das Objekt *box1* eine einzige n-wertige Variable ausreichend. Helmert [20] griff diese Idee auf, um Planungsprobleme mit ausschließlich Booleschen Variablen auf Planungsprobleme mit mehrwertigen Variablen zu verallgemeinern. Hierzu entwickelte er ein Verfahren, das für ein in PDDL 2.2 [13] kodiertes Planungsproblem Mengen von Zustandsvariablen bestimmt, so dass für alle Zustandsvariablen in derselben Menge zu jedem Zeitpunkt (das heißt in jedem Zustand) stets höchstens eine dieser Variablen wahr sein kann. Für jede dieser Mengen M kann nun eine $|M|$-wertige Variable angelegt werden.

ADDs [1] sind eine weitere Möglichkeit, die Heuristikwerte zu repräsentieren. Hierbei würde man für jedes Pattern P_i genau ein ADD berechnen, dessen Knoten den Zustandsvariablen aus Var^i und dessen Blätter den Heuristikwerten entsprechen.

In dieser Arbeit wurde eine von der Java Library bereit gestellte Hashtabelle verwendet, deren Speicherbedarf linear in der Anzahl der verwendeten Schlüssel und deren Zugriffsgeschwindigkeit im Optimalfall konstant ist.

5. Implementierung

In diesem Kapitel wird auf Praxisbezug und technische Details der Implementierung eingegangen.

5.1. Vorverarbeitung

Bevor die Pattern-Databases angelegt werden, findet ein Vorverarbeitungsschritt statt, der die Zustandsvariablen in zwei verschiedene Kategorien einteilt. Durch eine einfache syntaktische Analyse werden die Zustandsvariablen unterteilt in solche, die in jedem Zustand vorhanden sind, und in solche, für die diese Eigenschaft nicht gewährleistet werden kann. Die Menge der Zustandsvariablen, für die gewährleistet werden kann, dass sie in jedem Zustand vorhanden sind, ist diejenige Teilmenge des Initialzustands s_0, für die gilt, dass keine ihrer Zustandsvariablen als negativer Effekt in einer Aktion vorkommt. Dabei bezeichnet *negativer Effekt* eine Zustandsvariable aus der Menge *del* für $\langle add, del \rangle \in \mathit{eff}(a)$, wobei $a \in A$. Analog sei eine Zustandsvariable aus *add* ein *positiver Effekt*.
Diese Kategorisierung erlaubt es, die Patternselektion geschickter vorzunehmen. Es kann darauf verzichtet werden, Zustandsvariablen in ein Pattern aufzunehmen, die in jedem Zustand vorhanden sind, da hierdurch keinerlei zusätzliche Information gewonnen werden kann. Es wäre sinnvoller, statt dieser Zustandsvariablen eine andere in ein Pattern aufzunehmen, für die nicht gewährleistet ist, dass sie in jedem Zustand enthalten ist.
Alternativ wäre es auch möglich, die Zustandsvariablen, die in jedem Zustand enthalten sind, aus dem gesamten Planungsproblem zu entfernen.

5.2. Terminierung der Value-Iteration

Sei ein nichtdeterministisches Planungsproblem $\mathcal{P}$ gegeben, sowie ein Pattern P_i mit der entsprechenden Abstraktion $\mathcal{P}^i$. Sei $\mathcal{G}^i = (V^i, C^i)$ dessen vollständiger UND/ODER-Graph.

Da der Graph $\mathcal{G}^i$ Zustände mit unendlichem Heuristikwert enthalten kann, ist die Frage von Interesse, wie gewährleistet werden kann, dass die Value-Iteration angewendet auf $\mathcal{G}^i$, terminiert. Hierzu wird sich der folgenden Idee beholfen: Wegen der Korollare 4.4 und 4.8 brauchen alle abstrakten Zustände s^i nicht gespeichert zu werden, die entweder verloren oder nicht erreichbar sind, ohne dabei die Korrektheit zu gefährden, da auch alle nichtabstrakten Zustände s verloren sind, die diesen

Zuständen entsprechen (also Zustände $s \in S$ mit $s \cap P_i = s^i$).
Dies legt die Idee nahe, vor der Value-Iteration zunächst für jeden abstrakten Zustand zu berechnen, ob dieser gewonnen oder verloren ist, um alle verlorenen Zustände aus $\mathcal{G}^i$ zu entfernen, da für alle verbleibenden Zustände aufgrund des endlichen Heuristikwerts Terminierung des Value-Iteration-Algorithmus gewährleistet werden kann. Es bleibt folglich zu klären, wie berechnet wird, ob ein Zustand gewonnen oder verloren ist, und auf welche Art und Weise man verlorene Zustände aus V^i entfernen darf, ohne dass die Korrektheit gefährdet oder die Heuristikwerte anderer Zustände verändert werden.

Die Berechnung, ob Knoten gewonnen oder verloren sind, erfolgt auf dieselbe Weise, wie in der Rückwärtspropagierung des AO*-Algorithmus. Selbst Zykel gefährden hier nicht die Terminierung, da jeder Knoten maximal ein Mal auf gelöst gesetzt werden kann und nur in diesem Fall Propagierungen stattfinden.

Nun können alle verlorenen Zustände aus dem Graphen gelöscht werden. Wichtig dabei ist, für jeden gelöschten Zustand alle Konnektoren zu löschen, in denen dieser Zustand vorkommt. Das heißt falls $\mathcal{G}^i = (V^i, C^i)$ und $v \in V^i$ ist verloren, dann ist $\mathcal{G}^{i'} = (V^{i'}, C^{i'})$ mit $V^{i'} = V^i \backslash \{v\}$ und $C^{i'} = C^i \backslash \{ (s, \{s_1, \ldots, s_n\}) \in C^i \mid v = s \text{ oder } v \in \{s_1, \ldots, s_n\} \}$. Das Entfernen des gesamten Konnektors (statt beispielsweise lediglich einer Teilkante des entsprechenden Konnektors) ist erforderlich, um die Heuristikwerte nicht zu verfälschen.

Abbildung 5.1 zeigt zur Illustration einen vollständigen abstrakten UND/ODER-Graphen $\mathcal{G}^i$, der noch verlorene Zustände enthält (genau einen, nämlich s_1). Abbildung 5.2 zeigt die resultierenden Graphen, die entstehen, wenn man entweder nur die Kanten entfernt, in welchen verlorene Zustände auftreten (links in der Abbildung), oder falls man entsprechend den gesamten Konnektor entfernt (rechts in der Abbildung). Entfernt man nur die entsprechenden Kanten, so werden die Kostenwerte unter Umständen zu gering abgeschätzt. Entfernt man hingegen die gesamten Konnektoren, in welchen die verlorenen Zustände auftreten, so werden die Kosten hierdurch nicht beeinflusst. Dafür ist es aber möglich, dass der Graph in mehrere unzusammenhängende Teilgraphen zerfällt, wofür gegebenenfalls Vorkehrungen getroffen werden müssten.

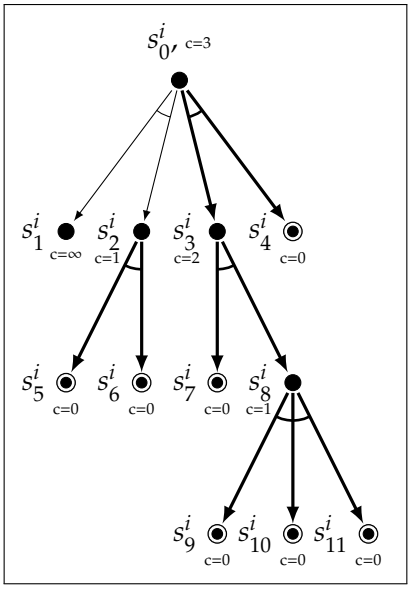
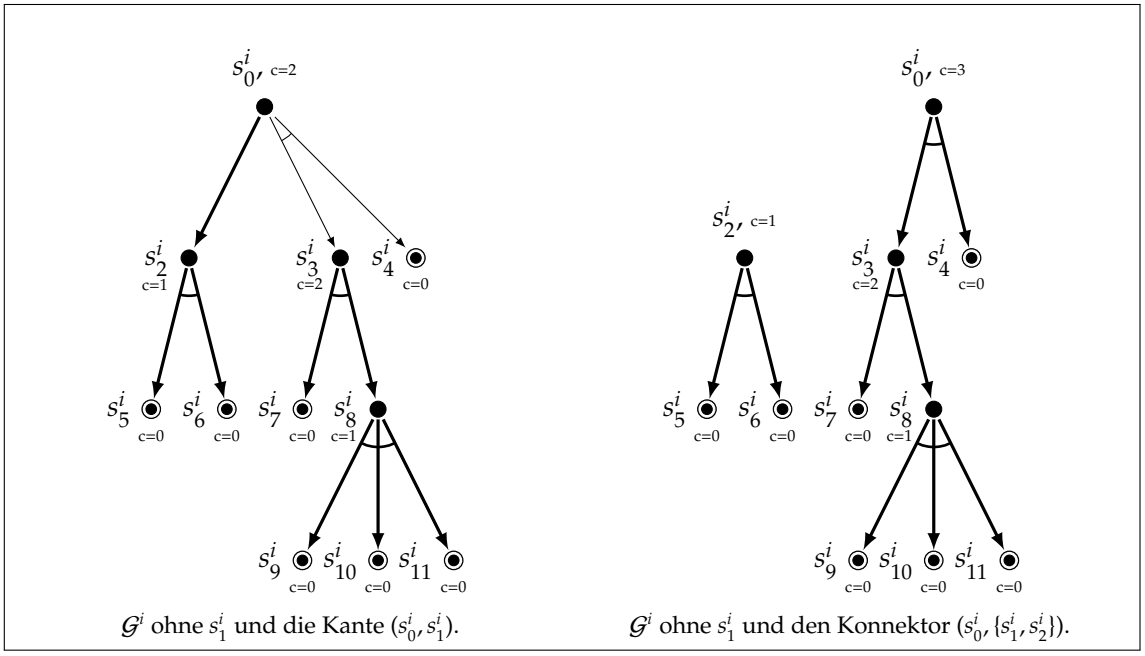


Abbildung 5.1.: Vollständiger abstrakter Graph $\mathcal{G}^i$.



$\mathcal{G}^i$ ohne s_1^i und die Kante (s_0^i, s_1^i).

$\mathcal{G}^i$ ohne s_1^i und den Konnektor $(s_0^i, \{s_1^i, s_2^i\})$.

Abbildung 5.2.: Vollständiger abstrakter Graph $\mathcal{G}^i$, aus welchem alle verlorenen Zustände und die entsprechenden Kanten bzw. Konnektoren entfernt wurden.

6. Experimente

Die in der vorliegenden Arbeit vorgestellte Technik wurde vollständig in Java implementiert und in verschiedenen Domänen mit anderen Heuristiken verglichen. Sämtliche Vergleiche beziehen sich damit auf dieselbe Implementierung, in der lediglich die verwendete Heuristik ausgetauscht wurde.
Als Suchalgorithmus wird eine Adaption des AO*-Algorithmus verwendet [2], welcher in der Lage ist, starke Pläne zu finden. Als Vergleichsreferenz zu der hier vorgestellten Pattern-Database-Heuristik dienen für ein nichtdeterministisches Problem $\mathcal{P}$ die beiden folgenden Heuristiken:

- Die *uninformierte Heuristik*, welche an noch zu expandierenden Knoten s stets den Heuristikwert 1 verwendet, sofern der Zustand s weder direkt gewonnen noch direkt verloren ist, das heißt falls $s \notin G$ und $\Gamma(s) \neq \emptyset$ (bzw. $C(s) \neq \emptyset$). Sie verwendet den Heuristikwert 0, falls s direkt gewonnen ist, das heißt falls $s \in G$, und ∞, falls s direkt verloren ist, das heißt falls $s \notin G$ und $\Gamma(s) = \emptyset$ (bzw. $C(s) = \emptyset$).

- Die *adversariale FF Heuristik* [3], welche durch Relaxierung des Planungsproblems für jeden Zustand s einen Pfad zu einem Zielzustand $s' \in G$ sucht und die Anzahl der verwendeten Aktionen dieses Pfades als Heuristikwert verwendet. Im Prinzip entspricht ein solcher Pfad einem relaxierten schwachen Plan. Die Relaxierung dieser Heuristik wird durch Ignorieren der negativen Effekte aller Aktionen erreicht. Das resultierende relaxierte Problem ist dadurch im Allgemeinen deutlich leichter zu lösen als das Originalproblem. Dennoch ist das *optimale* Lösen der relaxierten Variante NP-schwer [6]. Aus diesem Grund sucht diese Heuristik nicht nach einer optimalen Lösung, sondern liefert die erste zurück, die sie finden kann. Damit ist die adversariale FF Heuristik nicht zulässig.

Es folgt nun die Beschreibung der drei untersuchten Domänen, zusammen mit der Diskussion der experimentell ermittelten Ergebnisse. Die folgenden Domänen werden lediglich informell beschrieben, eine formale Domänenbeschreibung befindet sich im Anhang B.

Die Experimente wurden auf einem Computer mit Intel Core 2 Duo Prozessor mit 2,4GHz durchgeführt. Als Betriebssystem diente Windows Vista x64. Der Java-VM wurden 4GB Arbeitsspeicher reserviert. Für die Experimente gab es keine Laufzeitbeschränkung, lediglich der Arbeitsspeicher konnte unzureichend sein. Dies ist in den folgenden Tabellen durch ein X in der Spalte RAM kenntlich gemacht. Die Speicherangaben stellen nur eine Tendenz dar, da es unter Java nicht möglich ist, den augenblicklich belegten Speicher exakt zu bestimmen. Vor jeder Speicherabfrage wurde der Garbage Collector aufgerufen, der unter Java dafür verantwortlich

ist, nicht mehr referenzierte Objekte aus dem Speicher zu löschen. Dieser Aufruf garantiert aber nicht, dass der Garbage Collectors tatsächlich gestartet wird, er *empfiehlt* es lediglich. Dessen Start kann aus technischen Gründen nicht erzwungen werden.

In den folgenden Tabellen sind sämtliche Zeitangaben in Sekunden, die Angabe über den Speicherbedarf ist in Megabyte. Für jede betrachtete Heuristik werden fünf Spalten in die jeweiligen Tabellen aufgenommen:

- Z^1 ist die Zeit der Vorverarbeitung. In diese geht diejenige Zeit ein, welche vergeht, bevor der AO*-Algorithmus mit der Suche beginnt. Inbegriffen sind einfache Instantiierungen und Vorverarbeitungen, wie zum Beispiel in Abschnitt 5.1 beschrieben. Eine *bedeutsame* Vorverarbeitung findet nur bei den Pattern-Database-Heuristiken statt. Dort enthält Z^1 zusätzlich die Zeit, die benötigt wird, um die Pattern-Databases vollständig zu berechnen und zu speichern.
- Z^2 ist die Zeit, die zwischen dem Start und dem Ende des AO*-Algorithmus vergeht, inklusive sämtlicher Berechnungszeiten der Heuristikwerte.
- Z^3 ist schließlich die Summe der vorhergehenden Zeiten.
- *RAM* gibt den Gesamtspeicherbedarf an. Für die Pattern-Database-Heuristiken war es aus technischen Gründen nicht möglich, den Speicherbedarf der Pattern-Database vom restlichen belegten Speicher zu trennen.
- *Knoten* gibt die Zahl der während der Suche erstellen Knoten an. Hierbei ist folgende Anmerkung von Bedeutung:

Bei der Interpretation der nachfolgenden Ergebnisse ist zu beachten, dass die in den Tabellen aufgeführten expandierten Knoten nicht nur den Knoten entsprechen, die bislang behandelt wurden. In den bisher aufgeführten Beispielabbildungen von UND/ODER-Graphen entsprachen die Knoten genau den jeweiligen Zuständen, während die Aktionen über Konnektoren kodiert wurden. In der verwendeten Implementierung gibt es keine Konnektoren. Dafür wird für jede angewendete Aktion ein weiterer Knoten in den Graphen aufgenommen, die sogenannten UND-Knoten. Die *normalen* Knoten, die die Zustände aus S repräsentieren, heißen ODER-Knoten. Abbildung 6.1 stellt für einen Beispielgraphen beide möglichen Repräsentationen gegenüber, dabei entsprechen gefüllte Knoten den *normalen* Knoten, das heißt den ODER-Knoten und die leeren Knoten entsprechen den UND-Knoten, die die Konnektoren kodieren. Für eine detaillierte Beschreibung sei auf die Studienarbeit [2] verwiesen, aus welcher der verwendete Suchalgorithmus hervorging.

Die Verwendung von expliziten UND- und ODER-Knoten kann sowohl Einfluss auf die Komplexität der Laufzeit als auch auf die Komplexität der Anzahl der expandierten Knoten nehmen. Sofern die gewählte Repräsentation Einfluss auf die Komplexität der Laufzeit oder der Anzahl der expandierten Knoten nimmt, wird dies in der Diskussion der Ergebnisse hervorgehoben.

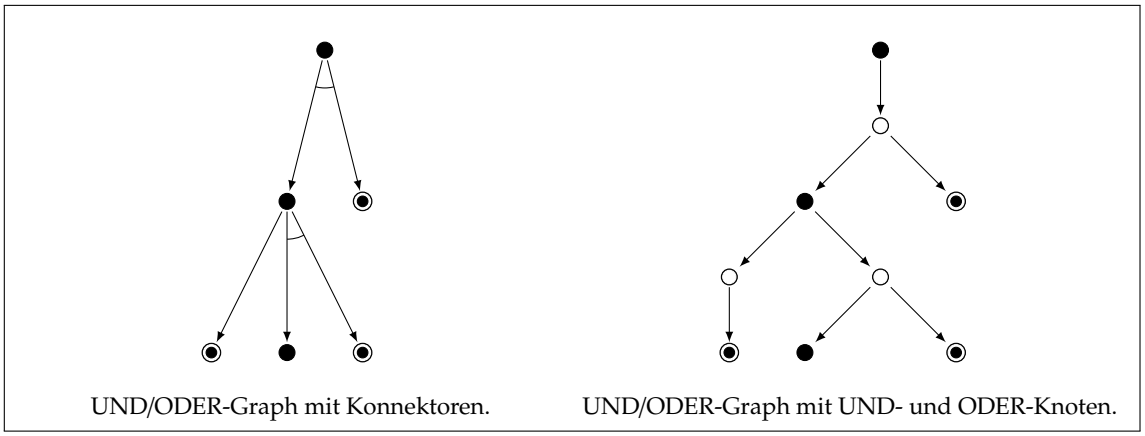


Abbildung 6.1.: Gegenüberstellung verschiedener Repräsentationen eines UND/ODER-Graphen.

6.1. Tireworld

Die hier vorgestellte Tireworld-Domäne ist eine modifizierte Variante der nichtdeterministischen Tireworld-Domäne der IPPC2008[1], welche wiederum eine Modifikation der deterministischen Tireworld-Domäne der IPC-6[2] darstellt.
Die Tireworld-Variante der IPPC2008 lässt lediglich starke zyklische Pläne zu, während die im Folgenden vorgestellte Variante auch starke Pläne zulässt. Von einer hierfür notwendigen Anpassung abgesehen sind beide Varianten identisch.

6.1.1. Problembeschreibung

Gegeben sind eine Menge von Orten und eine Menge von Straßen, die in beide Richtungen befahrbar sind und einige Orte miteinander verbinden. Die Aufgabe besteht darin, mit einem Auto von einem gegebenen Startort zu einem gegebenen Zielort zu gelangen. Hierfür kann das Auto nur die existierenden Straßen nutzen, wobei jede Straßenbenutzung nichtdeterministisch in einem platten Rad resultieren kann. Bevor das Auto von dem neuen Ort aus eine neue Straße befahren kann, muss dieses Rad ausgewechselt werden, wozu ein Ersatzrad benötigt wird. Das Auto, genau wie jeder Ort, kann maximal ein Ersatzrad lagern. Falls das Auto noch kein Ersatzrad geladen hat (initial besitzt das Auto kein Ersatzrad), kann es eines aufladen, sofern es sich an einem Ort befindet, welcher gerade ein Rad gelagert hat.
Die Aktionen dieses Problems können (informell) wie folgt ausgedrückt werden:

- Fahre von einem Ort zu einem angrenzenden Ort (das heißt die beiden Orte sind durch eine Straße verbunden), falls das Auto keinen Platten hat. Diese Aktion kann

[1]The Uncertainty Part of the 6th International Planning Competition 2008
[2]The 6th International Planning Competition

nichtdeterministisch darin resultieren, dass das Auto einen Platten bekommt. Unabhängig von diesem nichtdeterministischen Ausgang kommt das Auto stets an dem entsprechenden angrenzenden Ort an.

- Repariere das platte Rad, falls das Auto einen Platten hat und ein Ersatzrad besitzt. Nach der Reparatur besitzt das Auto kein Ersatzrad mehr.
- Lade ein Ersatzrad auf, falls das Auto noch keines besitzt und der aktuelle Ort ein solches gelagert hat. Nach dem Aufladen besitzt der Ort kein Ersatzrad mehr.

Repräsentiert man das Problem durch einen Graphen (hier Roadmap genannt) und betrachtet den Spezialfall, dass der Startort kein Ersatzrad gelagert hat, so kann man das Problem auch folgendermaßen ausdrücken: Gesucht ist ein Pfad vom Startknoten s zum Zielknoten s', dessen innere Knoten alle ein Ersatzrad gelagert haben. Weist man weiterhin allen Orten außer s und s' ein Ersatzrad zu, so liefert die optimale Lösung dieses Problems den kürzesten Pfad zwischen s und s'.

6.1.2. Beispiel

Abbildung 6.2 zeigt eine Instanz dieser Domäne mit den fünf Orten 1 bis 5. Die Straßen entsprechen den Kanten. Der Startort (Ort 1) ist durch einen eingehenden Pfeil gekennzeichnet, während der Zielort (Ort 5) durch eine zusätzliche Umrandung des Knotens gekennzeichnet ist. Jeder Ort, der ein Ersatzrad besitzt, ist durch eine graue Färbung des jeweiligen Knotens gekennzeichnet. In der durch diese Abbildung kodierten Domäne existieren folglich zwei starke Pläne. Der optimale Plan erfordert die Benutzung der Straße $\{1, 3\}$ und schließlich der Straße $\{3, 5\}$, während ein weiterer Plan den Umweg über Ort 4 kodiert.

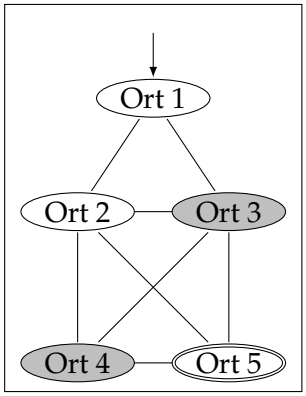


Abbildung 6.2.: Roadmap einer Tireworld-Domäne mit fünf Orten.

6.1.3. Diskussion der Ergebnisse

Es wurden mehrere Instanzen dieses Problems erstellt und mit den bereits vorgestellten Heuristiken verglichen. Es ist zu vermerken, dass die Suchzeit und die Anzahl der

expandierten Knoten der unterschiedlichen Heuristiken lediglich innerhalb einer Instanz miteinander vergleichbar sind, jedoch nicht zwischen den Instanzen. Dies geht auf den verwendeten Zufallsgenerator zurück, der nicht gewährleistet, dass ein Problem mit n_1 Orten einfacher zu lösen ist als ein Problem mit n_2 Orten und $n_1 < n_2$. Weiterhin ist zu vermerken, dass die Anzahl der Aktionen linear in der Anzahl der Orte steigt, da die Roadmap linear viele Straßen enthält und für jede Straße zwei Aktionen (für jede Fahrtrichtung eine) kodiert werden. Die Anzahl der Straßen hängt jedoch nicht in eindeutiger Weise von der Anzahl der Orte ab, sondern weiterhin vom Zufall.

Es werden nachfolgend die Ergebnisse von Tabelle 6.1 diskutiert.

Da es in dieser Domäne kein Skalierungsverhalten gibt, kann über die *uninformierte* und die *adversariale FF-Heuristik* wenig ausgesagt werden. Die bedeutendsten Erkenntnisse sind, dass die adversariale FF-Heuristik bis zu 17 mal weniger Knoten expandiert als die uninformierte Heuristik und alle Probleme lösen kann, während der uninformierten Heuristik bei zwei Problemen der Speicher ausgeht.

Für die Pattern-Database-Heuristiken wurden drei verschiedene Patternselektionsstrategien untersucht. Jede Heuristik greift auf zehn verschiedene Pattern-Databases zu. Jede einzelne dieser Pattern-Databases wurde mit einem Pattern, bestehend aus zehn Zustandsvariablen aufgebaut. Im Folgenden sei mit *Zustandsvariable eines Ortes* diejenige Zustandsvariable bezeichnet, die die Position des Autos an dem jeweiligen Ort kodiert. Da lediglich eine Zielzustandsvariable existiert (die des Zielorts), ist diese in allen Patterns enthalten. Die Wahl der restlichen neun Zustandsvariablen geht teilweise auf eine Zufallsselektion zurück. Die Selektionsstrategie z_1 selektiert zusätzlich die Zustandsvariable, die kodiert, ob das Auto einen Ersatzreifen bei sich führt. Weiterhin enthält sie die Zustandsvariable, die kodiert, ob das Auto einen Platten hat. Die restlichen sieben Zustandsvariablen entsprechen sieben zufällig gewählten Orten. Die Selektionsstrategie z_2 selektiert zusätzlich zum Zielort neun weitere Orte, während die Selektionsstrategie z_3 neun beliebige Zustandsvariablen selektiert. Die drei unterschiedlichen Selektionsstrategien, deren Ergebnisse in Tabelle 6.2 dargestellt sind, werden nach der Gegenüberstellung der verschiedenen Heuristiken betrachtet, die nun folgt. Für diesen Vergleich wird die Selektionsstrategie z_1 betrachtet, die durchschnittlich am besten abschneidet.

Die **Pattern-Database-Heuristiken**^{z_1} zeigen in der **Anzahl der expandierten Knoten** gerade in den kleinen Domänengrößen die besten Ergebnisse. Dies ist kein überraschendes Ergebnis, da aufgrund der geringen Anzahl an Zustandsvariablen die Abstraktionen eine nahezu perfekte Information liefern. Mit größer werdender Domäne ist die Anzahl der expandierten Knoten etwa mit denen der adversarialen FF-Heuristik vergleichbar.
Die **Laufzeit der Vorverarbeitung** der Heuristik ist linear in der Anzahl der Aktionen, da jede Aktion abstrahiert werden muss. Das Aufbauen des abstrakten Suchraums ist konstant, da dieser exponentiell durch die Patterngröße, also durch 2^{10}, beschränkt

ist und in jedem abstrakten Zustand alle abstrakten Aktionen auf Anwendbarkeit überprüft werden müssen, von welchen nach der Abstraktion nur noch konstant viele existieren. Die **Gesamtlaufzeit** betreffend schneidet diese Heuristik etwa gleich gut ab wie die adversariale FF-Heuristik.
Der **Speicherbedarf**, den diese Heuristik zusätzlich für die Speicherung der Pattern-Databases benötigt, ist minimal, was man besonders gut an den ersten Domänen erkennt. Somit kann der zusätzliche Speicherbedarf vernachlässigt werden.

Nachfolgend werden die Ergebnisse von Tabelle 6.2 diskutiert, in denen die verschiedenen Selektionsstrategien gegenübergestellt werden.

Insgesamt schneidet die bereits betrachtete Selektionsstrategie am besten ab. Überraschend ist jedoch, dass die Selektionsstrategie z_3 in allen Punkten mindestens so gut wie die Selektionsstrategie z_2 abschneidet. Das ist dadurch zu erklären, dass in mindestens einer der zehn Pattern-Databases, über die maximiert wurde, zufällig eine der Zustandsvariablen enthalten ist, die den Platten und den Besitz des Ersatzrads kodieren, die in jedem Pattern der Selektionsstrategie z_1 enthalten sind. Offenbar erhöht das Enthaltensein eines dieser Zustandsvariablen stark die Qualität der erhaltenen Heuristikwerte. Es wurden auch Experimente durchgeführt, in denen die Strategie z_3 dahingehend angepasst wurde, dass bei der zufälligen Selektion genau diese beiden Zustandsvariablen nicht in die Patterns aufgenommen werden konnten. In diesem Fall unterschieden sich die Ergebnisse nicht merklich von denen der Strategie z_2; daher wurden diese Ergebnisse nicht mehr explizit aufgeführt.

Besonders in dieser Domäne wäre die Verwendung mehrwertiger Variablen wünschenswert gewesen, da die Position des Autos jeweils über eine eigene Boolesche Zustandsvariable kodiert wird, obwohl das Auto höchstens an einem Ort gleichzeitig sein kann. Durch die Verwendung mehrwertiger Variablen hätten alle diese Zustandsvariablen in nur einer einzigen zusammengefasst werden können. Dadurch hätten bei vergleichbarem Speicherbedarf der Pattern-Databases weitere Variablen hinzugenommen, beziehungsweise größere Wertebereiche verwendet werden können, wodurch deutlich informativere Heuristikwerte hätten erzielt werden können.

Zusammenfassend erkennt man, dass die Pattern-Database-Heuristiken in der Anzahl der expandierten Knoten und in der Gesamtlaufzeit vergleichbare Ergebnisse wie die adversariale FF-Heuristik liefern. Man sieht aber weiterhin, insbesondere an den verschiedenen Selektionsstrategien, die in Tabelle 6.2 dargestellt sind, dass die Güte der Pattern-Database-Heuristiken sehr stark von der Wahl der Patterns abhängt.

#Orte	Uninformierte Heuristik					Adversariale FF-Heuristik					Pattern-Database-Heuristiken^{z_1}				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	0	**0**	**0**	**0**	85	0	**0**	**0**	**0**	62	0	**0**	**0**	**0**	**46**
6	0	**0**	**0**	**0**	84	0	**0**	**0**	**0**	47	2	**0**	2	8	**37**
8	0	**0**	**0**	1	153	0	**0**	**0**	**0**	**43**	1	**0**	1	6	**43**
10	0	**0**	**0**	1	293	0	**0**	**0**	**0**	96	2	**0**	2	7	**67**
20	0	**0**	**0**	4	504	0	**0**	**0**	**1**	**126**	2	**0**	2	8	128
40	0	93	93	2097	115541	0	**5**	**6**	**117**	**6451**	3	6	9	243	12827
60	0	2	2	196	9088	0	**0**	**1**	**24**	**1086**	2	**0**	2	41	1565
80	0	1	1	163	5924	0	**0**	**0**	**26**	915	1	**0**	2	30	**746**
100	0	17	17	745	20249	0	1	**1**	**42**	**1101**	1	**0**	2	50	1154
120	-	-	-	X	-	0	17	17	**464**	10739	1	**1**	**3**	81	**1632**
140	0	23	23	1584	33418	1	3	5	**90**	1849	2	**0**	**2**	66	**1202**
160	-	-	-	X	-	0	7	8	205	3914	2	**0**	**3**	**72**	**1175**
180	2	8	11	552	9599	0	**2**	**2**	**62**	**1055**	2	2	4	136	2239
200	1	39	41	1622	22175	0	4	5	130	1746	2	**1**	**3**	**93**	**1081**

Tabelle 6.1.: Ergebnisse der Tireworld-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^{z_1}: Maximierung über zehn Patterns mit Selektionsstrategie z_1:
Jedes Pattern enthält den Zielort, das Ersatzrad, den Platten und sieben zufällige Orte.

#Orte	Pattern-Database-Heuristiken^{z_1}					Pattern-Database-Heuristiken^{z_2}					Pattern-Database-Heuristiken^{z_3}				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	**0**	**0**	**0**	**0**	**46**	**0**	**0**	**0**	**0**	**46**	**0**	**0**	**0**	**0**	**46**
6	**2**	**0**	**2**	8	37	**2**	**0**	**2**	8	37	**2**	**0**	**2**	**5**	**28**
8	**1**	**0**	**1**	6	**43**	**1**	**0**	**1**	**5**	**43**	**1**	**0**	**1**	**5**	**43**
10	**2**	**0**	**2**	7	**67**	3	**0**	3	9	159	**2**	**0**	**2**	**6**	91
20	**2**	**0**	**2**	**8**	128	4	**0**	4	11	510	**2**	**0**	**2**	**8**	**112**
40	**3**	**6**	**9**	**243**	**12827**	4	94	99	2107	115607	4	7	11	302	16240
60	**2**	**0**	**2**	**41**	1565	5	3	8	203	9088	3	**0**	4	43	**1505**
80	**1**	**0**	**2**	**30**	**746**	4	2	6	171	5924	3	**0**	3	35	874
100	**1**	**0**	**2**	**50**	**1154**	3	22	26	750	20249	3	23	27	750	20249
120	**1**	**1**	**3**	**81**	**1632**	X	-	-	X	-	X	-	-	X	-
140	**2**	**0**	**2**	**66**	**1202**	7	54	62	1593	33347	4	47	52	1585	33347
160	**2**	**0**	**3**	**72**	1175	X	-	-	X	-	4	0	5	76	**1142**
180	**2**	**2**	**4**	**136**	**2239**	5	10	15	557	9599	4	14	18	543	9599
200	**2**	**1**	**3**	**93**	**1081**	4	48	52	1630	22166	4	**1**	5	98	1101

Tabelle 6.2.: Ergebnisse der Tireworld-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^{z_i}: Maximierung über zehn Patterns mit Selektionsstrategie z_i:
z_1: Zielort, Ersatzrad, Platten und sieben zufällige Orte.
z_2: Zielort und neun zufällige Orte.
z_3: Zielort und neun zufällige Zustandsvariablen.

6.2. Chain-of-Rooms

Die hier vorgestellte Chain-of-Rooms-Domäne ist eine Anpassung der Originalversion [30] und lässt sowohl starke zyklische Pläne als auch konformante Pläne zu.

6.2.1. Problembeschreibung

Gegeben ist eine Kette von Räumen, die nebeneinander angeordnet sind. Je zwei benachbarte Räume sind durch genau eine Tür verbunden, welche entweder geöffnet oder geschlossen sein kann. Es liegt hier also partielle Beobachtbarkeit vor, die als Nichtdeterminismus kodiert wird (jede Tür ist nach einer Observationsaktion nichtdeterministisch entweder geöffnet oder geschlossen). Ein Agent befindet sich zu Beginn im ersten Raum und hat die Aufgabe, alle Räume zu besuchen. Um von einem Raum zum nächsten zu gelangen, muss die sie verbindende Tür benutzt werden, welche gegebenenfalls erst geöffnet werden muss. Bevor allerdings eine Tür durchquert oder geöffnet werden kann, muss zunächst im jeweiligen Raum das Licht angeschaltet werden, damit der Agent die Tür sichtet. Initial sind in allen Räumen die Lichter ausgeschaltet.
Die Aktionen dieses Problems können (informell) wie folgt ausgedrückt werden:

- Schalte das Licht in dem Raum an, in welchem sich der Agent gerade befindet, sofern dieses noch aus ist. Diese Aktion resultiert in dem Wahrnehmen der Tür des Raumes. Die Tür kann nichtdeterministisch geöffnet oder geschlossen sein.
- Öffne die Tür des Raumes, in dem sich der Agent gerade befindet, sofern er diese wahrnehmen kann (das heißt falls das Licht angeschaltet ist). Diese Aktion ist auch ausführbar, falls die Tür bereits geöffnet ist.
- Verlasse den aktuellen Raum in den benachbarten Raum, falls die sie verbindende Tür gesehen werden kann und diese geöffnet ist. Ein Raum kann somit in beide benachbarten Räume verlassen werden (sofern alle Bedingungen erfüllt sind), auch wenn es keine Notwendigkeit gibt, in bereits besuchte Räume zurückzukehren, da sich der Agent zu Beginn im ersten Zimmer befindet und daher direkt alle Räume durchqueren kann, ohne einen Raum mehrmals zu besuchen.

6.2.2. Beispiel

Sei eine Kette von $n+1$ Räumen gegeben, das heißt es existieren n Türen, die durchquert werden müssen.

Der gesuchte starke Plan ist offensichtlich: Schalte im ersten Raum das Licht an. Falls die Tür geöffnet ist, durchquere sie. Falls die Tür geschlossen ist, öffne sie und durchquere sie danach. Diese Vorgehensweise wird insgesamt n mal durchgeführt, bis der letzte Raum erreicht und damit alle Räume besucht wurden.

Der kürzeste konformante Plan besteht aus der n-maligen Wiederholung der Aktionssequenz Licht anschalten, Tür öffnen und Tür durchqueren.

6.2.3. Diskussion der Ergebnisse

In dieser Domäne ist der erreichbare Zustandsraum quadratisch in der Anzahl der Räume. Dies liegt daran, dass der Agent beim Betreten eines neuen Raumes stets wieder bis zum ersten Raum zurück laufen kann. Betritt der Agent zum ersten Mal einen Raum i und schreitet sofort zurück zu Raum $i-1$, so können diese beiden Zustände, die die Anwesenheit in Raum $i-1$ kodieren, nicht miteinander identifiziert werden, da durch das erstmalige Betreten des Raumes i neue Information gewonnen wird. Durch die so entstehenden neuen Zustände ergibt sich eine Menge von erreichbaren Zuständen, deren Größe quadratisch in der Anzahl der Räume ist.

Es folgt eine Diskussion über die ermittelten Werte aus Tabelle 6.3.

Die **Anzahl der expandierten Knoten** der **uninformierten Heuristik** ist quadratisch in der Anzahl der Räume, was der Intuition entspricht, dass die uninformierte Heuristik den gesamten Zustandsraum expandiert. Die **Laufzeit** und der **Speicherbedarf** sind aufgrund der vielen expandierten Knoten den anderen Heuristiken stets unterlegen.

Die **Anzahl der expandierten Knoten** der **adversarialen FF-Heuristik** ist stets etwa halb so groß wie die Anzahl der durch die uninformierte Heuristik expandierten Knoten. Folglich expandiert auch diese Heuristik quadratisch viele Knoten. Der konstante Faktor 2 lässt sich dadurch erklären, dass die Heuristikwerte der adversarialen FF-Heuristik nicht informativ genug sind, um die Suche geeignet zu leiten. Zu diesem Zweck betrachten wir Abbildung 6.3.

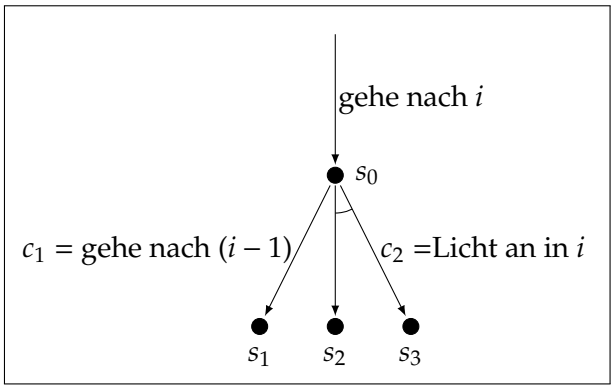


Abbildung 6.3.: Erklärung der Knotenexpansionen.

Wird das erste Mal ein neuer Raum betreten, in diesem Fall Raum i, so sind zwei Aktionen anwendbar: *Zurück gehen* (es wurde bereits erörtert, dass dabei ein neuer Zustand generiert wird), repräsentiert durch den Konnektor c_1 und *Licht anschalten*,

repräsentiert durch den Konnektor c_2. Die Heuristikwerte der adversarialen FF-Heuristik sind nun dahingehend wie erwünscht, dass die Kosten des Zustands s_1 höher sind als die der Zustände s_2 und s_3. Folglich wird zunächst wie erwünscht s_2 oder s_3 expandiert. Da die Heuristik nicht perfekt ist und und sich weiterhin die Heuristikwerte der Zustände s_1, s_2 und s_3 nur sehr wenig unterscheiden, übersteigen irgendwann aufgrund der Propagierungen die Kostenwerte von s_2 und s_3 den Heuristikwert von s_1. Dadurch wird der Knoten s_1 expandiert, das heißt der Agent läuft wieder zurück. Dies geschieht so lange, bis entweder die Kosten von s_1 so weit heraufgesetzt werden, dass der Konnektor c_2 markiert wird, oder bis der erste Raum erreicht wurde und daher im optimalen Teillösungsgraphen keine Knoten mehr zur Expansion existieren. Auch in diesem Fall geht die Markierung von c_1 auf c_2 über. Dass quadratisch viele Knoten erstellt werden, aber dennoch deutlich weniger als unter Verwendung der uninformierten Heuristik, lässt sich wie folgt erklären. Gegen Ende des Suchvorgangs, also wenn sich der Agent nur noch wenige Räume vor dem letzten Raum befindet, ist die Heuristik genau genug, damit keine unerwünschten Knoten mehr expandiert werden. Je tiefer im Suchgraphen diese Situation eintritt, desto größer ist die Ersparnis der expandierten Knoten.
Es bleibt die Diskussion der **Laufzeit**. Das starke Wachstum der Suchzeit ist durch die Vorgehensweise der adversarialen FF-Heuristik zu begründen. In dieser Domäne ist ihre Laufzeit pro Knoten quadratisch in Abhängigkeit der Anzahl der Räume und ist damit insgesamt in $O(n^4)$ für n = Anzahl der Räume. Um die quadratische Laufzeit der adversarialen FF-Heuristik zu verstehen, betrachten wir (stark vereinfacht) ihre Vorgehensweise: In jedem Zustand werden alle existierenden relaxierten Aktionen (von welchen $O(n)$ viele existieren) auf Anwendbarkeit überprüft. Alle anwendbaren Aktionen werden gleichzeitig angewendet, wodurch *ein* neuer Zustand erstellt wird, der alle positiven Effekte aller angewendeten Aktionen enthält. Diese Vorgehensweise wird wiederholt, bis ein Zustand generiert wird, der alle Zielzustandsvariablen enthält. Da die Aktionen nur sequentiell anwendbar sind, ergibt sich eine Pfadlänge, die linear in der Anzahl der Räume ist. Da für jeden Zustand auf diesem Pfad $O(n)$ viele Aktionen überprüft und gegebenenfalls angewendet werden müssen, ergibt sich eine quadratische Laufzeit.

Für die Pattern-Database-Heuristiken wurden Experimente mit sechs verschiedenen Patternselektionen durchgeführt. Jede dieser Selektionen gewährleistet Additivität. In dieser Domäne ist die Idee zum Erhalt additiver Patterns, die Zustandsvariablen benachbarter Räume in dasselbe Pattern zu übernehmen. Das Gesamtproblem wird somit in mehrere kleine Teilprobleme aufgeteilt. Die Zustandsvariablen sollen auf eine Weise auf die Patterns verteilt werden, dass Satz 4.2 angewendet werden kann, der Additivität gewährleistet. Aufgrund des Kriteriums des eindeutigen Besitzers kann der Satz nur dann angewendet werden, wenn die Zerlegung der Räume so gewählt wird, dass zwischen je zwei angrenzenden Folgen von Räumen ein Raum freigelassen wird. Betrachten wir als Beispiel ein Problem mit acht Räumen. Angenommen, es werden zwei Patterns P_1 und P_2 gewählt, so dass die Zustandsvariablen der Räume 1 bis 4 in Pattern P_1 enthalten sind und entsprechend die Zustandsvariablen der Räume 5 bis 8 in

Pattern P_2. Auf diese Zerlegung ist Satz 4.2 nicht anwendbar, da die Aktionen *gehe von 4 nach 5* und *gehe von 5 nach 4* keinen eindeutigen Besitzer haben. Die Eindeutigkeit des Besitzers ist verletzt, da beide Aktionen offensichtlich Zustandsvariablen aus beiden Patterns enthalten müssen. Wird hingegen das Problem so zerlegt, dass jeweils ein Raum freigelassen wird, besteht diese Problematik nicht und alle Aktionen haben einen eindeutigen Besitzer. Allerdings stellt Satz 4.2 lediglich ein *hinreichendes* Kriterium für Additivität dar, kein *notwendiges*. In dieser Domäne ist nun selbst dann Additivität gegeben, wenn kein Raum freigelassen wird. Auf einen formalen Beweis dieser Aussage wird an dieser Stelle verzichtet. Da beide Patternselektionssystematiken (Raum freilassen, Raum nicht freilassen) Additivität gewährleisten, wurden Experimente mit beiden Systematiken durchgeführt. Dabei wurden Patterns gewählt, die jeweils alle Zustandsvariablen von drei, vier oder fünf benachbarten Räumen beinhalten. Die Ergebnisse der verschiedenen Patternselektionen werden später diskutiert. Zunächst werden die Ergebnisse der uninformierten und der adversarialen FF-Heuristik mit denen der Pattern-Database-Heuristiken^{4n} verglichen, die das Problem in Teilprobleme mit jeweils *vier* Räumen aufteilen, die *n*icht durch einen weiteren Raum separiert sind.

Auffällig bei der **Anzahl der expandierten Knoten** der **Pattern-Database-Heuristiken**^{4n} ist, dass sie stets nur etwas geringer ist als bei der adversarialen FF-Heuristik. Dies ist dadurch zu begründen, dass die Pattern-Database-Heuristiken dasselbe Problem wie die adversarialen FF-Heuristik haben, das durch Abbildung 6.3 illustriert wurde: Sie sind nicht genügend informativ, aber etwas informativer als die adversariale FF-Heuristik.
Die **Laufzeit der Vorverarbeitung** ist quadratisch in der Anzahl der Räume. Es existieren O(n) viele Patterns und für jedes müssen alle Abstraktionen der Aktionen gebildet werden, von welchen O(n) viele existieren. Dennoch ist der Hauptgrund für den hohen Anstieg der Vorverarbeitungszeit die gewählte Repräsentation mit expliziten UND- und ODER-Knoten. Diese führt dazu, dass nach der Abstraktion noch immer O(n) viele Aktionen übrig bleiben. Bei einer Repräsentation mit Konnektoren wären nach der Abstraktion nur noch konstant viele Aktionen übrig geblieben. Alle weiteren würden aufgrund des eindeutigen Besitzers der Aktionen die Form $\langle\emptyset, \{\langle\emptyset, \emptyset\rangle\}\rangle$ annehmen. Auf weitere Details sei an dieser Stelle verzichtet.
Obwohl Pattern-Database-Heuristiken zusätzlichen **Speicherbedarf** für die Pattern-Databases haben, fällt dieser so gering aus, dass der Gesamtverbrauch aufgrund der geringeren Knotenexpansionen stets kleiner ist als bei den anderen Heuristiken.
Insgesamt ist diese Heuristik entweder gleich gut wie die Vergleichsheuristiken oder sie ist ihnen überlegen.

Nun werden die Ergebnisse der Problempartitionierungen in (nicht)separierte Folgen von drei, vier und fünf Räumen diskutiert, die in den Tabellen 6.4 und 6.5 zusammengefasst sind.

Beiden Tabellen ist gemeinsam, dass die Anzahl der expandierten Knoten sinkt, je mehr Räume die Patterns beinhalten. Dieser Sachverhalt ist trivial, da mit steigender

Anzahl der Räume pro Pattern auch der Informationsgehalt der Heuristik steigt und damit weniger Knoten expandiert werden müssen. Ebenfalls erkennt man, dass die Dauer des Vorverarbeitungsschritts sehr stark wächst, je mehr Räume ein Pattern enthält. Auch dies ist zu erwarten, da der abstrakte Zustandsraum, der in der Vorverarbeitung exploriert wird, oft seine maximale Größe annimmt, da auch Zielzustände weiter expandiert werden. Es ist also erkennbar, dass größere Patterns zwar weniger Knotenexpansionen bewirken, allerdings beeinträchtigt dies gleichzeitig in hohem Maße die Vorverarbeitungszeit. Es ist daher eine nichttriviale Frage, wie groß die Patterns zu wählen sind, um eine optimale Gesamtsuchzeit zu erreichen. Zwischen den Tabellen 6.4 und 6.5 gibt es keine nennenswerten Unterschiede. Die Ergebnisse der Pattern-Database-Heuristiken ohne Separation sind deutlich besser als die der Pattern-Database-Heuristiken mit Separation. Da es sich bei beiden Heuristiken um additive Heuristiken handelt, war dieses Ergebnis zu erwarten, da ohne Separation mehr Patterns in die Berechnung eingehen können und dadurch bessere Heuristikwerte erzielt werden.

#Räume	Uninformierte Heuristik					Adversariale FF-Heuristik					Pattern-Database-Heuristiken^{4n}				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	0	**0**	**0**	**0**	38	0	**0**	**0**	**0**	**30**	0	**0**	**0**	**0**	**30**
6	0	**0**	**0**	**0**	91	0	**0**	**0**	**0**	62	0	**0**	**0**	**0**	**57**
8	0	**0**	**0**	1	170	0	**0**	**0**	**0**	109	0	**0**	**0**	**0**	**81**
10	0	**0**	**0**	**1**	272	0	**0**	**0**	**1**	169	0	**0**	**0**	**1**	**147**
20	0	**0**	**0**	6	1142	0	**0**	**0**	**3**	643	1	**0**	1	**3**	**464**
40	0	1	**1**	40	4682	0	8	8	20	2494	3	**0**	4	**18**	**2043**
60	0	5	**5**	121	10621	0	40	40	59	5543	9	**1**	10	**52**	**4827**
80	0	17	**17**	278	18962	0	122	122	131	9793	16	**3**	20	**121**	**8804**
100	0	43	43	533	29702	0	300	300	251	15244	30	**9**	**39**	**230**	**13983**
120	0	92	92	909	42841	0	609	609	443	21893	43	**19**	**62**	**394**	**20367**
140	0	162	162	1390	58382	0	1164	1164	686	29743	63	**37**	**101**	**617**	**27944**
160	0	257	257	2090	76322	0	2053	2054	994	38794	94	**63**	**157**	**902**	**36723**
180	0	468	468	2986	96661	0	3369	3369	1457	49043	127	**98**	**225**	**1283**	**46707**
200	-	-	-	X	-	0	5855	5855	1961	60493	164	**154**	**319**	**1763**	**57884**

Tabelle 6.3.: Ergebnisse der Chain-of-Rooms-Domäne. Vergleich der verschiedenen Heuristiken.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^{4n}: Vier *n*ichtseparierte Räume pro Pattern

#Räume	Pattern-Database-Heuristiken^{3n}					Pattern-Database-Heuristiken^{4n}					Pattern-Database-Heuristiken^{5n}				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	**0**	**0**	**0**	**0**	33	**0**	**0**	**0**	**0**	**30**	**0**	**0**	**0**	**0**	**30**
6	**0**	**0**	**0**	**0**	**57**	**0**	**0**	**0**	**0**	**57**	**0**	**0**	**0**	**0**	**57**
8	**0**	**0**	**0**	1	95	**0**	**0**	**0**	**0**	**81**	**0**	**0**	**0**	1	**81**
10	**0**	**0**	**0**	1	171	**0**	**0**	**0**	1	147	**0**	**0**	**0**	**0**	**105**
20	**0**	**0**	**0**	**3**	639	1	**0**	1	**3**	464	2	**0**	2	**3**	**361**
40	**1**	**0**	**1**	26	2823	3	**0**	4	18	2043	14	**0**	14	**17**	**1577**
60	**3**	1	**4**	90	5995	9	1	10	**52**	4827	35	**0**	36	59	**3808**
80	**6**	5	**11**	241	11127	16	3	20	121	8804	65	**2**	68	**95**	**7085**
100	**10**	13	**24**	297	17847	30	9	39	**230**	13983	107	**6**	114	280	**11376**
120	**16**	28	**45**	483	24907	43	19	62	394	20367	174	**14**	188	**315**	**16717**
140	**22**	55	**78**	767	34575	63	37	101	617	27944	241	**25**	267	**496**	**23076**
160	**32**	97	**130**	1154	45831	94	63	157	902	36723	344	**45**	390	**738**	**30461**
180	**44**	151	**195**	1594	56779	127	98	225	1283	46707	457	**75**	533	**1052**	**38877**
200	**59**	237	**297**	2217	70983	164	154	319	1763	57884	580	**111**	692	**1443**	**48308**

Tabelle 6.4.: Ergebnisse der Chain-of-Rooms-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^{in}: *i n*ichtseparierte Räume pro Pattern.

#Räume	Pattern-Database-Heuristiken^{3s}					Pattern-Database-Heuristiken^{4s}					Pattern-Database-Heuristiken^{5s}				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	**0**	**0**	**0**	**0**	33	**0**	**0**	**0**	**0**	**30**	**0**	**0**	**0**	**0**	**30**
6	**0**	**0**	**0**	**0**	74	**0**	**0**	**0**	**0**	88	**0**	**0**	**0**	**0**	**57**
8	**0**	**0**	**0**	1	118	**0**	**0**	**0**	**0**	**112**	**0**	**0**	**0**	1	125
10	**0**	**0**	**0**	**1**	194	**0**	**0**	**0**	**1**	167	**0**	**0**	**0**	**1**	**149**
20	**0**	**0**	**0**	5	896	**0**	**0**	**0**	**4**	763	1	**0**	1	5	**668**
40	**1**	1	**2**	32	3781	3	**0**	4	27	3305	8	**0**	9	**22**	**2641**
60	**3**	4	**7**	100	8657	8	2	10	**84**	7647	22	**1**	24	97	**6796**
80	**5**	10	**15**	223	15538	15	8	23	**195**	13789	38	**4**	43	267	**12267**
100	**8**	27	**36**	429	24424	24	19	44	372	21731	63	**12**	76	**313**	**18650**
120	**13**	56	**70**	716	35316	36	47	84	632	31473	99	**27**	126	**560**	**28256**
140	**19**	112	132	1116	48195	51	71	**123**	981	43015	139	**50**	189	**870**	**38603**
160	**27**	179	**206**	1660	63072	76	132	208	1460	56357	192	**81**	274	**1253**	**49373**
180	**37**	274	311	2377	79955	104	201	**305**	2080	71499	276	**175**	452	**1847**	**64459**
200	**43**	902	945	3221	97302	133	375	**509**	2859	88441	443	**269**	713	**2531**	**79626**

Tabelle 6.5.: Ergebnisse der Chain-of-Rooms-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^{is}: *i s*eparierte Räume pro Pattern.

6.3. Coin-Flip

Die Coin-Flip-Domäne wurde für diese Arbeit entwickelt, da sie sich aufgrund vieler unabhängiger Teilprobleme besonders gut für Pattern-Database-Heuristiken eignet.

6.3.1. Problembeschreibung

Gegeben ist eine feste Anzahl von Münzen, die initial alle auf ihrer Seite stehen. Das Ziel ist es, dass irgendwann alle Münzen Kopf zeigen. Zu diesem Zweck kann eine noch stehende Münze geworfen werden, worauf die Münze nichtdeterministisch Kopf oder Zahl zeigt. Jede liegende Münze kann umgedreht werden, damit ihre entsprechend andere Seite nach oben zeigt.
Die Aktionen dieses Problems können (informell) wie folgt ausgedrückt werden:

- Werfe eine noch stehende Münze. Diese Aktion resultiert nichtdeterministisch in dem Zeigen von Kopf oder Zahl der entsprechenden Münze.
- Drehe eine Münze um. Zeigte sie Kopf, zeigt zeigt sie nun Zahl; zeigte sie Zahl, zeigt sie nun Kopf.

6.3.2. Beispiel

Es seien n Münzen gegeben, die initial alle auf ihrer Seite stehen.

Jeder optimale starke Plan enthält für jede Münze genau die Aktionen *Münze werfen* und *Münze umdrehen*. Die Reihenfolge, in welcher diese Aktionen vorgenommen werden ist dabei beliebig, wobei eine Münze erst umgedreht werden kann, nachdem sie geworfen wurde.

Folglich hat jeder optimale Plan die Länge $2n$. Ein möglicher Plan wäre (informell) *Münze 1 werfen*, gegebenenfalls *Münze 1 umdrehen*,. . . ,*Münze n werfen* und schließlich gegebenenfalls *Münze n umdrehen*.

6.3.3. Diskussion der Ergebnisse

In dieser Domäne ist der erreichbare Zustandsraum exponentiell in Anzahl der Münzen. Dies ist offensichtlich, da sich bei n Münzen n unabhängige Teilprobleme ergeben, die in beliebiger Reihenfolge gelöst werden können. Darüber hinaus müssen die Teilprobleme auch nicht erst vollständig gelöst werden, bevor ein anderes Teilproblem angegangen wird. So kann man zum Beispiel die erste Münze werfen, danach eine weitere und dann erst die erste Münze auf die richtige Seite umdrehen (sofern dies erforderlich ist).
Es ist zu erwarten, dass in dieser Domäne sämtliche zulässige, nichtperfekte Heuristiken im Suchraum verloren gehen. Dies liegt daran, dass Zustände, die sich sehr tief

im Suchgraphen und entsprechend nahe an einem Zielzustand befinden, durch die Rückwärtspropagierungen bewirken, dass es irgendwann günstiger sein wird, einen Zustand zu expandieren, der näher beim Initialzustand des Planungsproblems liegt. Für eine genauere Ausführung dieser Idee sei auf die Arbeit *How Good is Almost Perfect?* [21] verwiesen.

Im Folgenden werden die Ergebnisse von Tabelle 6.6 diskutiert.

Zur **uninformierten Heuristik** und zur **adversarialen FF-Heuristik** gibt es nicht viel zu sagen: Beide gehen wie erwartet im Suchraum verloren. Was hingegen auffällt, insbesondere im Vergleich zur vorangegangenen Chain-of-Rooms-Domäne, ist die besonders geringe Laufzeit der adversarialen FF-Heuristik. Dies liegt daran, dass ihre Laufzeit in dieser Domäne linear in Anzahl der Münzen ist, da in jedem Zustand stets alle Aktionen anwendbar sind und somit nach deren Anwendung bereits ein Zielzustand hergeleitet werden kann.

In dieser Domäne ist man in der Lage, additive Patterns zu finden, die den Bedingungen von Satz 4.2 genügen. Hierzu müssen alle Zustandsvariablen, die zur selben Münze gehören, in ein eigenes Pattern. Für n Münzen kann man also n Patterns erstellen, die Additivität gewährleisten. Für die Eindeutigkeit des Besitzers spielt es dabei keine Rolle, ob die Patterns nur die Zustandsvariablen einer oder mehrerer Münzen beinhalten. Daher wurde mit Patterns experimentiert, die eine, zwei oder drei Münzen beinhalten. Die Diskussion dieser unterschiedlichen Pattern-Database-Heuristiken folgt später. Zunächst werden die Ergebnisse der uninformierten und der adversarialen FF-Heuristik denen der Pattern-Database-Heuristiken[2] gegenübergestellt, die die Zustandsvariablen von zwei Münzen pro Pattern enthält.

Das wichtigste Ergebnis ist, dass sämtliche Pattern-Database-Heuristiken perfekt sind, dass diese also c^* exakt berechnen. Durch die völlige Unabhängigkeit der Teilprobleme kann durch die Addition ihrer Heuristikwerte der Abstand zu einem Ziel exakt bestimmt werden. Folglich ist auch die **Anzahl der expandierten Knoten** aller **Pattern-Database-Heuristiken** minimal, unabhängig von der Patterngröße. Das heißt es wird kein einziger Zustand expandiert, der nicht expandiert werden muss. Obwohl für ein Problem mit n Münzen maximal $O(n)$ viele Aktionen ausgeführt werden müssen, steigt die Anzahl der expandierten Knoten quadratisch. Dabei spielt es keine Rolle, welche Repräsentation der UND/ODER-Graphen gewählt wird. Für die Repräsentation mit UND- und ODER-Knoten kann die Anzahl der expandierten Knoten durch die Formel $\frac{21}{5} + 5 * (-\frac{2}{5} + n)^2$ ermittelt werden (hergeleitet durch das Lösen eines Gleichungssystems auf Grundlage der experimentellen Ergebnisse). Die quadratische Anzahl der Knoten ergibt sich daraus, dass in jedem Zustand alle Aktionen angewendet werden müssen, um die Heuristiken der Kinder zu betrachten. Anfangs sind noch $O(n)$ viele Aktionen anwendbar, dann $O(n-1)$ viele, bis schließlich keine mehr anwendbar sind.
Die **Laufzeit der Vorverarbeitung** ist quadratisch in Abhängigkeit der Anzahl der

Münzen, da bei zwei Münzen pro Pattern $\frac{n}{2} \in O(n)$ viele Patterns existieren und für jedes $O(n)$ viele Aktionen abstrahiert werden müssen. Der Hauptgrund für den hohen Anstieg der Vorverarbeitungszeit ist allerdings, genau wie in der Chain-of-Rooms-Domäne, die gewählte Repräsentation mit expliziten UND- und ODER-Knoten, die dazu führt, dass auch nach der Abstraktion $O(n)$ viele Aktionen übrig bleiben, statt konstant viele. Die **Laufzeit der Suche** ist quadratisch in Anzahl der expandierten Knoten. Zu erwarten wäre lediglich eine lineare Laufzeit in Anzahl der expandierten Knoten, da aufgrund der perfekten Information keine Rückwärtspropagierungen stattfinden können. Damit ergibt sich für jeden expandierten Knoten eine lineare Laufzeit, da $O(n)$ viele Aktionen auf Anwendbarkeit überprüft werden müssen. Auch hier geht die quadratische Laufzeit auf die gewählte Repräsentation zurück.

Nachfolgend werden die Ergebnisse diskutiert, wenn die Pattern-Database-Heuristiken mit den Zustandsvariablen von einer, zwei oder drei Münzen pro Pattern erstellt wurden. Die Ergebnisse sind in Tabelle 6.3.3 zusammenfasst.

Wie bereits erörtert, nimmt es auf den Informationsgehalt der Pattern-Database-Heuristiken keinen Einfluss, ob die Patterns die Zustandsvariablen von einer, zwei oder drei Münzen enthalten, womit sich auch die **Anzahl der expandierten Knoten** nicht unterscheiden. Auf die **Laufzeit** nimmt die Größe der Patterns dahingehend Einfluss, dass die Vorverarbeitungszeit für jede weitere Münze sehr stark zunimmt, da der Zustandsraum exponentiell in Größe des Patterns wächst. Gleichzeitig sinkt die reine Suchzeit, obwohl die Anzahl der expandierten Knoten gleich bleibt. Die Ursache ist, dass sich der Heuristikwert eines nichtabstrakten Zustands aus der Summe der Heuristikwerte seiner Abstraktionen ergibt. Befinden sich in jedem Pattern die Zustandsvariablen von i Münzen, so müssen pro Zustand $\frac{n}{i}$ Berechnungen durchgeführt werden, falls n = Anzahl der Münzen. Da die Laufzeit der Vorverarbeitung sehr stark wächst, ist es sinnvoll die Anzahl der Münzen pro Pattern in Relation zur Anzahl der expandierten Knoten zu erhöhen. Falls sehr viele Knoten erstellt werden, relativiert die Einsparung beim Zugriff auf den Heuristikwert eines Zustands die hohe Dauer des Vorverarbeitungsschritts.

Zusammenfassend können in dieser Domäne die Pattern-Database-Heuristiken als klar überlegen bezeichnet werden, da sie durch Ausnutzung der Additivität und durch die Unabhängigkeit der Teilprobleme die exakte Kostenfunktion c^* berechnen.

#Münzen	Uninformierte Heuristik					Adversariale FF-Heuristik					Pattern-Database-Heuristiken^2				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	0	**0**	**0**	1	349	0	**0**	**0**	1	313	0	**0**	**0**	**0**	**69**
6	0	**0**	**0**	11	4809	0	**0**	**0**	8	3374	0	**0**	**0**	**0**	**161**
8	0	18	18	145	58377	0	3	3	46	18641	0	**0**	**0**	**1**	**293**
10	0	1102	1102	1542	588813	0	77	77	483	186221	0	**0**	**0**	**1**	**465**
20	-	-	-	X	-	-	-	-	X	-	0	**0**	**0**	**7**	**1925**
40	-	-	-	X	-	-	-	-	X	-	1	**1**	**2**	**49**	**7845**
60	-	-	-	X	-	-	-	-	X	-	3	**6**	**9**	**161**	**17765**
80	-	-	-	X	-	-	-	-	X	-	6	**18**	**24**	**384**	**31685**
100	-	-	-	X	-	-	-	-	X	-	12	**45**	**57**	**714**	**49605**
120	-	-	-	X	-	-	-	-	X	-	18	**96**	**114**	**1176**	**71525**
140	-	-	-	X	-	-	-	-	X	-	28	**163**	**192**	**1784**	**97445**
160	-	-	-	X	-	-	-	-	X	-	39	**289**	**329**	**2589**	**127365**
180	-	-	-	X	-	-	-	-	X	-	58	-	-	X	-
200	-	-	-	X	-	-	-	-	X	-	74	-	-	X	-

Tabelle 6.6.: Ergebnisse der Coin-Flip-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
Pattern-Database-Heuristiken^2: Zwei Münzen pro Pattern.

#Münzen	Pattern-Database-Heuristiken[1]					Pattern-Database-Heuristiken[2]					Pattern-Database-Heuristiken[3]				
	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten	Z^1	Z^2	Z^3	RAM	Knoten
4	**0**	**0**	**0**	**0**	**69**	**0**	**0**	**0**	**0**	**69**	**0**	**0**	**0**	**0**	**69**
6	**0**	**0**	**0**	**0**	**161**	**0**	**0**	**0**	**0**	**161**	**0**	**0**	**0**	**0**	**161**
8	**0**	**0**	**0**	**1**	**293**	**0**	**0**	**0**	**1**	**293**	**0**	**0**	**0**	**1**	**293**
10	**0**	**0**	**0**	**1**	**465**	**0**	**0**	**0**	**1**	**465**	**0**	**0**	**0**	**1**	**465**
20	**0**	**0**	**0**	8	**1925**	**0**	**0**	**0**	**7**	**1925**	**0**	**0**	**0**	8	**1925**
40	**0**	2	**2**	**49**	**7845**	1	**1**	**2**	**49**	**7845**	3	**1**	4	50	**7845**
60	**1**	9	11	161	**17765**	3	6	**9**	161	**17765**	9	**4**	14	**160**	**17765**
80	**3**	27	30	**347**	**31685**	6	18	**24**	384	**31685**	19	**13**	32	**347**	**31685**
100	**6**	83	89	720	**49605**	12	45	**57**	**714**	**49605**	37	**35**	72	**714**	**49605**
120	**9**	170	179	**1173**	**71525**	18	**96**	**114**	1176	**71525**	74	97	172	1179	**71525**
140	**13**	280	293	1786	**97445**	28	163	**192**	**1784**	**97445**	123	**158**	282	1796	**97445**
160	**18**	496	515	2594	**127365**	39	289	**329**	2589	**127365**	215	**249**	465	**2579**	**127365**
180	**24**	-	-	X	-	58	-	-	X	-	215	-	-	X	-
200	**34**	-	-	X	-	74	-	-	X	-	266	-	-	X	-

Tabelle 6.7.: Ergebnisse der Coin-Flip-Domäne.
Z^1: Zeit für die Vorverarbeitung, Z^2: Zeit für die Suche, Z^3: Gesamtzeit. Angaben in Sekunden.
PDB^i: *i* Münzen pro Pattern

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung

Ziel dieser Arbeit war es, Pattern-Database-Heuristiken, die nach Wissen des Autoren bislang nur im klassischen Planen eingesetzt wurden, auf domänenunabhängiges, nichtdeterministisches Planen zu übertragen und diese theoretisch und experimentell zu untersuchen.

Dazu wurden zunächst nichtdeterministische Planungsprobleme allgemein definiert. Weiterhin wurden UND/ODER-Graphen vorgestellt, die eine mögliche Repräsentation solcher Probleme darstellen. Gesucht ist ein starker Plan für nichtdeterministische Planungsprobleme, also eine Strategie, die unabhängig vom Ausgang des Nichtdeterminismus das Erreichen des Ziels garantiert. Um einen starken Plan mit dem Ansatz der heuristischen Suche zu finden, wurde der AO*-Algorithmus eingesetzt, der einen UND/ODER-Graphen aufbaut, um in diesem nach einem starken Plan zu suchen. Die Knoten dieses Graphen entsprechen den Zuständen des Planungsproblems. Um die Suche zu lenken, werden an den zu expandierenden Knoten Heuristiken eingesetzt.

Hierzu wurden Pattern-Database-Heuristiken vorgestellt, deren Funktionsweise es ist, vor der eigentlichen Suche die Heuristikwerte vorzuberechnen, um auf diese während der Laufzeit in möglichst kurzer Zeit zugreifen zu können. Diese Vorberechnung erfolgt in mehreren Schritten: Für ein gegebenes Pattern, also eine Teilmenge der Zustandsvariablen, wird das Planungsproblem abstrahiert. Abstraktion bedeutet dabei, dass das Planungsproblem vollständig auf das Pattern eingeschränkt wird, das heißt der Initialzustand, die Zielzustände und sämtliche Aktionen bestehen nur noch aus den Zustandsvariablen des Patterns. Dieses abstrahierte, also vereinfachte, Planungsproblem wird vollständig gelöst. Dabei wird für jeden im abstrakten Suchraum erreichbaren Zustand dessen optimaler Kostenwert berechnet. Die erreichbaren Zustände werden mit ihrem Kostenwert in der so genannten Pattern-Database abgelegt. Es werden mehrere dieser Abstraktionen gebildet, so dass auch mehrere Pattern-Databases existieren. Während der Suche wird zum Erhalt eines Heuristikwerts überprüft, welchen abstrakten Zuständen in den Pattern-Databases jeweils der Zustand entspricht, für den ein Heuristikwert berechnet werden soll. Aus allen Pattern-Databases wird der entsprechende Kostenwert abgerufen, um durch Maximierung oder Addition dieser Werte den gewünschten Heuristikwert zu berechnen. Die auf diese Weise konstruierte Heuristik ist zulässig. Dass die konstruierte Heuristik trotz Addition zulässig ist, geht auf Satz 4.2 zurück, der das wichtigste Ergebnis dieser Arbeit darstellt und ein allgemeines Kriterium nennt, wann Addition verwendet werden kann.

Das gesamte Verfahren wurde in Java implementiert. Die experimentell ermittelten Ergebnisse konnten zeigen, dass Pattern-Database-Heuristiken auch im nichtdeterministischen Planen großes Potential haben. Die besten Ergebnisse konnten dabei erzielt werden, wenn viele unabhängige Teilprobleme vorliegen, da hierdurch Additivität besonders gut ausgenutzt werden kann.

7.2. Ausblick

Der wichtigste noch offene Punkt ist die automatisierte Patternselektion. Die Pattern-Database-Heuristiken wurden als domänenunabhängige Heuristiken vorgestellt. Folglich ist es auch zwingend erforderlich, die Patternselektion automatisiert vorzunehmen, da bei einer handkodierten, domänenspezifischen Patternselektion nicht von Domänenunabhängigkeit gesprochen werden kann. Während im klassischen Planen bereits Arbeiten zur automatisierten Patternselektion [17, 12, 18] vorliegen, ist die Adaption auf nichtdeterministisches Planen noch nicht umgesetzt.

Ein weiterer, äußerst wichtiger Punkt ist die Umstellung der Repräsentation der UND/ODER-Graphen auf Graphen mit Konnektoren, statt mit UND- und ODER-Knoten. Hierdurch kann je nach Domäne sowohl die Laufzeit als auch der Speicherbedarf drastisch reduziert werden. Mit der Umstellung der Repräsentation geht auch die Änderung des Suchalgorithmus einher. Dabei ist eine Anpassung wünschenswert, die neben starken Plänen auch starke zyklische und konformante Pläne finden kann. Weiterhin kann die augenblicklich vorliegende Implementierung zwar mit Zyklen umgehen, dabei wird aber nicht gewährleistet, dass die Kosten der Knoten stets den wünschenswerten Wert annehmen. Dort besteht daher noch Verbesserungsbedarf.

Der letzte große Punkt ist die Erweiterung auf mehrwertige Variablen. Hierfür gibt es zwei mögliche Ansätze: Das Eingabeformat kann dahingehend angepasst werden, dass einzelnen Variablen explizit ein Wertebereich zugeteilt wird. Alternativ kann das aktuelle Eingabeformat, welches auf Booleschen Variablen arbeitet, beibehalten werden und stattdessen ein weiterer Vorverarbeitungsschritt eingebaut werden, der aus den Booleschen Variablen eine mehrwertige Kodierung gewinnt. Im klassischen Planen ist diese Variante die übliche Vorgehensweise. Die hierbei verwendete Systematik [20] müsste noch auf nichtdeterministisches Planen erweitert werden.

Literaturverzeichnis

[1] Bahar, Ruth Iris, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo und Fabio Somenzi: *Algebraic Decision Diagrams and their Applications*. In: *International Conference on Computer-Aided Design, IEEE*, Seiten 188–191, 1993.

[2] Bercher, Pascal: *Lösen rundenbasierter Erreichbarkeitsspiele unter Verwendung des AO*-Algorithmus und einer Relaxierungsheuristik*. Studienarbeit, Albert-Ludwigs-Universität Freiburg im Breisgau, 2008.

[3] Bercher, Pascal und Robert Mattmüller: *A Planning Graph Heuristic for Forward-Chaining Adversarial Planning*. In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, Seiten 921–922, 2008.

[4] Bonet, Blai und Hector Geffner: *Planning as Heuristic Search*. Artificial Intelligence, 129(1–2):5–33, 2001.

[5] Bryant, Randal E.: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 35:677–691, 1986.

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

[7] Cimatti, Alessandro, Marco Pistore, Marco Roveri und Paolo Traverso: *Weak, strong, and strong cyclic planning via symbolic model checking*. Artificial Intelligence, 147(1–2):35–84, 2003.

[8] Culberson, Joseph C. und Jonathan Schaeffer: *Searching with Pattern Databases*. In: *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, Seiten 402–416. Springer-Verlag, 1996.

[9] Culberson, Joseph C. und Jonathan Schaeffer: *Pattern Databases*. Computational Intelligence, 14(3):318–334, 1998.

[10] Edelkamp, Stefan: *Planning with Pattern Databases*. In: *Proceedings of the 6th European Conference on Planning (ECP-01)*, Seiten 13–24, 2001.

[11] Edelkamp, Stefan: *Symbolic Pattern Databases in Heuristic Search Planning*. In: *Proceedings of AIPS-02*, Seiten 274–283, 2002.

[12] Edelkamp, Stefan: *Automated Creation of Pattern Database Search Heuristics*. In: *MoChArt*, Seiten 35–50, 2006.

[13] Edelkamp, Stefan und Jörg Hoffmann: *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition*. Technischer Bericht 238, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.

[14] Felner, Ariel, Richard Korf und Sarit Hanan: *Additive Pattern Database Heuristics*. Journal of Artificial Intelligence Research, 22:279–318, 2004.

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

[16] Fox, Maria und Derek Long: *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. Journal of Artificial Intelligence Research, 20:61–124, 2003.

[17] Haslum, Patrick, Blai Bonet und Héctor Geffner: *New Admissible Heuristics for Domain-Independent Planning*. In: *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, Band 20, Seiten 1163–1168, 2005.

[18] Haslum, Patrik, Adi Botea, Blai Bonet, Malte Helmert und Sven Koenig: *Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning*. In: *In Proc. AAAI 2007*, Seiten 1007–1012, 2007.

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

[20] Helmert, Malte: *Concise finite-domain representations for PDDL planning tasks*. Artificial Intelligence, 2009 (in Druck).

[21] Helmert, Malte und Gabriele Röger: *How Good is Almost Perfect?* In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, Seiten 944–949, 2008.

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

[23] Kautz, Henry und Bart Selman: *Planning as satisfiability*. In: *Proceedings of the 10th European conference on Artificial intelligence*, Seiten 359–363, 1992.

[24] Kautz, Henry und Bart Selman: *Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search*. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, Seiten 1194–1201. AAAI Press, 1996.

[25] Korf, Richard E. und Ariel Felner: *Disjoint Pattern Database Heuristics*. Artificial Intelligence Journal (AIJ), 134:9–22, 2002.

[26] Loyd, Sam und Martin Gardner: *Mathematical Puzzles*. Courier Dover Publications, 1959.

[27] Nilsson, Nils J.: *Problem-Solving Methods in Artificial Intelligence*, Kapitel 5: Problem-Reduction Search Methods, Seiten 116–155. McGraw-Hill Inc., 1971.

[28] Nilsson, Nils J.: *Principles of Artificial Intelligence*, Kapitel 3: Search Strategies for Decomposable Production Systems, Seiten 99–129. Springer-Verlag, 1982.

[29] Prieditis, Armand E.: *Machine Discovery of Effective Admissible Heuristics*. In: *Machine Learning*, Seiten 117–141, 1993.

[30] Rintanen, Jussi: *Constructing Conditional Plans by a Theorem-Prover*. Journal of Artificial Intelligence Research, 10:323–352, 1999.

[31] Sadikov, Aleksander und Ivan Bratko: *Pessimistic Heuristics Beat Optimistic Ones in Real-Time Search*. In: *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, Seiten 148–152, 2006.

[32] Samadi, Mehdi, Jonathan Schaeffer, Fatemeh Torabi Asr, Majid Samar und Zohreh Azimifar: *Using Abstraction in Two-Player Games*. In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, Seiten 545–549, 2008.

[33] Schaeffer, Jonathan, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu und Steve Sutphen: *Checkers Is Solved*. Science, 317(5844):1518–1522, 2007.

[34] Sutton, Richard S. und Andrew G. Barto: *Reinforcement Learning: An Introduction*. MIT Press, 1998.

A. Abbildungsverzeichnis

2.1. Vollständiger UND/ODER-Graph von $\mathcal{P}$. 7
2.2. Sämtliche Lösungsgraphen von $\mathcal{P}$. 8
2.3. Ein UND/ODER-Graph von $\mathcal{P}$, der keinen Lösungsgraphen darstellt. . . 9

3.1. Beispiel zum AO*-Algorithmus (1). 14
3.2. Beispiel zum AO*-Algorithmus (2). 15

4.1. Kommutierendes Diagramm. 20
4.2. Vollständiger UND/ODER-Graph von $\mathcal{P}$. 22
4.3. Die vollständigen UND/ODER-Graphen der Abstraktionen $\mathcal{P}^1, \mathcal{P}^2, \mathcal{P}^3$. . 24
4.4. Konnektor mit unterschiedlichen Besitzern und dessen Abstraktionen. . 26
4.5. Lösungsgraph von $\mathcal{P}$. 33
4.6. Abstrakte Lösungsgraphen für die Patterns P_1 und P_2. 35

5.1. Vollständiger abstrakter Graph $\mathcal{G}^i$. 43
5.2. $\mathcal{G}^i$ nach Entfernen verlorener Zustände. 43

6.1. Verschiedene Repräsentationen eines UND/ODER-Graphen. 47
6.2. Roadmap einer Tireworld-Domäne mit fünf Orten. 48
6.3. Erklärung der Knotenexpansionen. 54

B. Formale Problembeschreibungen

Im Folgenden werden die NuPDDL[1]-Kodierungen der experimentell untersuchten Domänen vorgestellt. Diese Kodierungen können von dem Planer, der in dieser Arbeit verwendet wurde, *nicht* gelesen werden. Sie dienen lediglich der Illustration der verwendeten Domänen. NuPDDL ist die Eingabesprache des Planers MBP [7], die eine Erweiterung der Eingabesprache PDDL2.1 [16] darstellt.

B.1. Tireworld-Kodierungen

Die Tireworld-Domäne und die Tireworld-Problembeschreibung kodieren das Beispiel 6.1.2 mit fünf Orten.

```
(define (domain tireworld)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location
                     ?to - location)
               (flattire)
               (hasspare)
  )

  (:action move-car
   :parameters (?from - location ?to - location)
   :precondition (and (vehicle-at ?from)
                      (road ?from ?to)
                      (not (flattire))
                 )
```

Problembeschreibung 1: Tireworld-Domäne (Teil 1/2)

[1] http://sra.itc.it/tools/mbp/index.html#nupddl

```
   :effect (and (vehicle-at ?to)
                (not (vehicle-at ?from))
                (oneof (and)
                       (flattire)
                )
           )
  )

  (:action loadtire
   :parameters (?loc - location)
   :precondition (and (vehicle-at ?loc)
                      (spare-in ?loc)
                      (not (hasspare))
                 )
   :effect (and (hasspare)
                (not (spare-in ?loc))
           )
  )

  ;; Angepasste Aktion 'changetire', die auch
  ;; starke Pläne erlaubt.
  (:action changetire
   :precondition (hasspare)
   :effect (and (not (hasspare))
                (not (flattire))
           )
  )

;; Originalaktion 'changetire' der ippc2008, die
;; lediglich starke zyklische Pläne zulässt, da die
;; Aktion nichtdeterministisch misslingen könnte.
;; (:action changetire
;;  :precondition (hasspare)
;;  :effect (oneof (and)
;;                 (and (not (hasspare))
;;                      (not (flattire))
;;                 )
;;          )
;; )

)
```

Problembeschreibung 2: Tireworld-Domäne (Teil 2/2)

```
(define (problem tireworld)
  (:domain tireworld)
  (:objects n1 n2 n3 n4 n5 - location)

  (:init (vehicle-at n1)
         (road n1 n2) (road n2 n1)
         (road n1 n3) (road n3 n1)
         (road n2 n3) (road n3 n2)
         (road n2 n4) (road n4 n2)
         (road n2 n5) (road n5 n2)
         (road n3 n4) (road n4 n3)
         (road n3 n5) (road n5 n3)
         (road n4 n5) (road n5 n4)
         (spare-in n3)
         (spare-in n4)
         (flattire)
  )

  (:goal (vehicle-at n5))

)
```

Problembeschreibung 3: Tireworld-Problem.

B.2. Chain-of-Rooms-Kodierungen

Die Chain-of-Rooms-Domäne und die Chain-of-Rooms-Problembeschreibung zeigen eine Instanz mit fünf Räumen.

```
(define (domain chainOfRooms)
  (:requirements :typing :strips :non-deterministic)
  (:types room)
  (:predicates (door_unlocked ?r - room)
               (visited ?r - room)
               (light_on ?r - room)
               (light_off ?r - room)
               (agent_position ?r - room)
               (adjacent ?rl - room ?rr - room)
  )
```

Problembeschreibung 4: Chain-of-Rooms-Domäne (Teil 1/2).

```
  (:action move_left_right
   :parameters (?rl - room ?rr - room)
   :precondition (and (agent_position ?rl)
                      (adjacent ?rl ?rr)
                      (door_unlocked ?rl)
                 )
   :effect (and (not (agent_position ?rl))
                (agent_position ?rr)
                (visited ?rr)
           )
  )

  (:action move_right_left
   :parameters (?rl - room ?rr - room)
   :precondition (and (agent_position ?rr)
                      (adjacent ?rl ?rr)
                      (door_unlocked ?rl)
                 )
   :effect (and (not (agent_position ?rr))
                (agent_position ?rl)
                (visited ?rl)
           )
  )

  (:action turn_light_on
   :parameters (?r - room)
   :precondition (and (light_off ?r)
                      (agent_position ?r)
                 )
   :effect (and (light_on ?r)
                (not (light_off ?r))
                (oneof (door_unlocked ?r)
                       (and )
                )
           )
  )

  (:action unlock_door
   :parameters (?r - room)
   :precondition (and (agent_position ?r)
                      (light_on ?r)
                 )
   :effect (door_unlocked ?r)
  )
)
```

Problembeschreibung 5: Chain-of-Rooms-Domäne (Teil 2/2).

```
(define (problem chainOfRooms)
  (:domain chainOfRooms)
  (:objects r1 r2 r3 r4 r5 - room)
  (:init (agent_position r1)
         (visited r1)
         (light_off r1)
         (light_off r2)
         (light_off r3)
         (light_off r4)
         (light_off r5)
         (adjacent r1 r2)
         (adjacent r2 r3)
         (adjacent r3 r4)
         (adjacent r4 r5)
  )
  (:goal (and (visited r1)
              (visited r2)
              (visited r3)
              (visited r4)
              (visited r5)
         )
  )
)
```

Problembeschreibung 6: Chain-of-Rooms-Problem.

B.3. Coin-Flip-Kodierungen

Die Coin-Flip-Domäne und die Coin-Flip-Problembeschreibung zeigen eine Instanz mit fünf Münzen.

```
(define (domain coinFlip)
  (:requirements :typing :strips :non-deterministic)
  (:types coin)
  (:predicates (stands ?c - coin)
               (heads ?c - coin)
               (tails ?c - coin)
  )
```

Problembeschreibung 7: Coin-Flip-Domäne (Teil 1/2).

```
  (:action coin-Flip
   :parameters (?c - coin)
   :precondition (stands ?c)
   :effect (and (not (stands ?c))
                (oneof (tails ?c)
                       (heads ?c)
                )
           )
  )

  (:action turn-tail-to-head
   :parameters (?c - coin)
   :precondition (tails ?c)
   :effect (and (not (tails ?c))
                (heads ?c)
           )
  )

  (:action turn-head-to-tail
   :parameters (?c - coin)
   :precondition (heads ?c)
   :effect (and (not (heads ?c))
                (tails ?c)
           )
  )
)
```

Problembeschreibung 8: Coin-Flip-Domäne (Teil 2/2).

```
(define (problem coinFlip)
  (:domain coinFlip)
  (:objects coin1 coin2 coin3 coin4 coin5 - coin)

  (:init (stands coin1)
         (stands coin2)
         (stands coin3)
         (stands coin4)
         (stands coin5)
  )

  (:goal (and (heads coin1)
              (heads coin2)
              (heads coin3)
              (heads coin4)
              (heads coin5)
         )
  )
)
```

Problembeschreibung 9: CoinFlip-Problem.