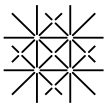


Universität
Basel

Berechnung von Potential-Heuristiken basierend auf Pattern-Datenbanken

Bachelorarbeit

Philosophisch-Naturwissenschaftliche Fakultät der Universität Basel
Department Mathematik und Informatik
Künstliche Intelligenz
http://ai.cs.unibas.ch

Beurteiler: Prof. Dr. Malte Helmert
Zweitbeurteiler: Thomas Keller und Salomé Eriksson

Andreas Ferenczi
andreas.ferenczi@stud.unibas.ch
2002-921-013

21. Dezember 2016

Danksagung

Ich möchte hier meinen Dank aussprechen an alle die mich in jeglicher Form unterstützt haben während der Anfertigung dieser Arbeit.

An erster Stelle gebührt mein Dank Prof.Dr. Malte Helmert. Er ermöglichte mir ein Thema zu bearbeiten, das ich mir nicht spannender hätte wünschen können. So nah am innersten Kern einer künstlichen Intelligenz zu arbeiten, hat mir so viel Freude bereitet, dass ich die damit verbundenen Anstrengungen kaum gefühlt habe.

Ebenfalls möchte ich mich bei Dr. Thomas Keller und Salomé Eriksson bedanken, für die vielen Ratschläge, Einweisungen, die Motivation und ganz besonders für die vielen Stunden, die sie für mich aufgewendet haben. Ohne ihre hervorragende Betreuung wäre diese Arbeit wohl gar nicht zustande gekommen.

Ausserdem muss ich mich bei meinem Vater bedanken, dafür dass er mit seiner Erfahrung in der Programmierung, wohl etliche mühselige Stunden Arbeit erspart hat, speziell im Debugging.

Auch meiner Mutter möchte ich danken, insbesondere für das Korrekturlesen, sowie für die stilistische Beratung während der Abfassung dieser Arbeit.

Ich möchte darüber hinaus einen speziellen Dank an meine Eltern richten, dafür dass sie mich während diesem Studium in vielerlei Hinsicht unterstützt haben und weiter unterstützen.

Die Experimente wurden auf sciCORE (http://scicore.unibas.ch) scientific comuting core facility an der Universität Basel durchgeführt und ich möchte meine Anerkennung für sciCORE und seine Mitarbeiter aussprechen.

Schliesslich gebührt auch Dank an meine Freunde Daniel Vosseberg, Ludwig Zumthor und Tobias Buser, die mir immer wieder Möglichkeit gegeben haben abzuschalten und frische Motivation tanken liessen.

Abstrakt

In dieser Arbeit wird versucht eine Heuristik zu lernen. Damit eine Heuristik erlernbar ist, muss sie über Parameter verfügen, die die Heuristik bestimmen. Eine solche Möglichkeit bieten *Potential-Heuristiken* und ihre Parameter werden *Potentiale* genannt.

Pattern-Databases können mit vergleichsweise wenig Aufwand Eigenschaften eines Zustandsraumes erkennen und können somit eingesetzt werden als Grundlage um Potentiale zu lernen.

Diese Arbeit untersucht zwei verschiedene Ansätze zum Erlernen der *Potentiale* aufgrund der Information aus *Pattern-Databases*. In Experimenten werden die beiden Ansätze genauer untersucht und schliesslich mit der FF-Heuristik verglichen.

Inhaltsverzeichnis

1 Einleitung

In *Planungsproblemen* geht es darum eine Folge von *Aktionen* (*Plan*) zu finden, die die Welt vom Status-Quo (*Initial-Zustand*) in einen gewünschten Zustand (*Zielzustand*) überführt. Um ein solches Planungsproblem lösen zu können, kann niemals die gesamte Welt berücksichtigt werden und es ist deshalb notwendig die Welt auf die für das Problem relevanten Variablen zu reduzieren. Um einen Rechner für ein derartiges Problem einsetzen zu können, müssen ausserdem alle Variablen digital erfassbar sein. Dies ermöglicht ohnehin schon nur eine beschränkte Anzahl möglicher Werte einer Variablen, darüber hinaus ist es sinnvoll die Anzahl möglicher Werte minimal zu halten, um so die Komplexität des Problems klein zu halten. Diese reduzierte Sicht der Welt nennt man den *Zustandsraum*. Er besteht aus *Zuständen* die jeweils eine spezifische Variablenbelegung repräsentieren.

Eine Folge von Aktionen muss also gefunden werden, um ein solches Problem innerhalb eines Zustandsraumes zu lösen. Hierbei hat jede Aktion einen Vorgängerzustand und einen Folgezustand. Um in einem gegebenen Vorgängerzustand eine Aktion auswählen zu können, müssen die Folgezustände miteinander verglichen werden bezüglich ihrer Distanz zum nächsten Zielzustand. Ein Mass für diese Distanz nennt man eine *Heuristik*. Falls diese Distanz exakt wiedergegeben wird, so nennt man dies die h^**-Heuristik*. In der Regel lässt sich dies jedoch nicht mit vernünftigem Aufwand genau berechnen und deshalb versuchen alle anderen Heuristiken eine Näherungslösung zu finden.

Um eine Heuristik lernen zu können, muss sie über Parameter verfügen, die sie komplett bestimmen. Eine solche Heuristik ist die *Potential-Heuristik* und ihre Parameter nennt man *Potentiale*. Die dabei verwendete Technik besteht darin, dass zu einigen Features je ein Potential gehört. Besitzt ein gegebener Zustand das Feature, so wird ihr Potential zu einer Summe hinzu gezählt, die schliesslich das Resultat dieser Heuristik (*h-Wert*) ist.

Im einfachsten Fall definiert man die Features als Fakten, bezihungsweise blosse Variablen-Werte-Paare. Als wahr erachtet man das Fakt, falls die gegebene Variable den gegebenen Wert hat, ansonsten ist das Fakt falsch. Zum Beispiel ist das Fakt $\mathcal{F}(x, 3)$ ausschliesslich dann wahr, wenn $x = 3$ gilt.

Auf jeden Fall braucht man aber eine Grundlage, worauf man diese Potentiale lernen möchte. *Pattern-Databases* reduzieren das Problem, indem sie bloss wenige Variablen berücksichtigen. So lässt sich das reduzierte Problem ohne eine Heuristik (sog. blinde Suche) lösen. Als *Pattern* der Pattern-Database bezeichnet man diese Auswahl von Variablen. Da nur Pattern-Variablen berücksichtigt wurden, sind die Distanzen zum Ziel in der Pattern-Database höchstens gleich gross wie h^* und bilden somit eine Schätzung dafür. Weil es viele mögliche Pattern-Databases zu einem Problem gibt und jede eine Schätzung für h-Werte generiert, eignen sie sich besonders gut für einen solchen Machine-Learning Ansatz.

In dieser Arbeit wurden zwei verschiedene Methoden zum Bestimmen der Potentiale einer Potential-Heuristik aufgrund von Pattern-Databases erarbeitet und untereinander sowie mit der gängigen *FF-Heuristik* verglichen.

2
Grundlagen

2.1 Definitionen

Eine **Variable** v ist ein Platzhalter für einen Wert $x \in \mathbb{G}$, wobei $\mathbb{G}$ eine Menge von Zahlen ist und als **Domäne** der Variablen bezeichnet wird.

Ein **Zustand** ist eine vollständige Belegung der Variablen des Problems und lässt sich als Vektor (**Zustandsvektor**) schreiben:
Sei $V = \{v_0, v_1, \ldots, v_{n-1}\}$ die Menge aller Variablen, sei $s = (x_0, x_1, \ldots, x_{n-1}) \in S$, so gilt im Zustand s: $v_i = x_i \forall i \in \mathbb{N}_0$ mit $i < n$. Mit S wird die Menge aller Zustände respektive, wo das passend ist, der Vektorraum der Zustandsvektoren bezeichnet.

Eine **Aktion** kann als 2-Tupel geschrieben werden:

$$a = \langle \mathcal{P}, \mathcal{E} \rangle$$

Hierbei ist $\mathcal{P}$ eine Menge von Bedingungen, die in einem Zustand erfüllt sein müssen, damit die Aktion **zulässig** ist. Ferner bezeichnet $\mathcal{E}$ die Menge der Effekte der Aktion, also Bedingungen die der **Folgezustand** der Aktion erfüllt. Die Aktion überführt den Zustand in ihren Folgezustand, wenn sie ausgeführt wird.

Eine **Planungsaufgabe** über ein 5-Tupel wie folgt:

$$\mathcal{D} = \langle V, A, \text{cost}, s_0, S_* \rangle$$

Hierbei ist:

- V die endliche Menge aller Variablen,
- A die endliche Menge aller Aktionen,
- $\text{cost} : A \to \mathbb{R}_0^+$ eine Funktion für die Kosten aller Aktionen,
- s_0 der Initial-Zustand und
- S_* die endliche Menge aller Zielzustände.

Ein **Pfad** $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ ist eine Sequenz von Aktionen, für die in einem gegebenen Zustand s_j gilt: $\langle s_{j+k}, a_k, s_{j+k+1} \rangle \in T \; \forall k \in \mathbb{N}_0$ mit $k \leq n$.

Ein **Plan** ist ein Pfad, der in einen Zielzustand überführt.

Ein **Suchalgorithmus** sucht nach einem Plan für eine gegebene Planungsaufgabe.

Eine **Heuristik** $h(s)$ ist eine Funktion $S \xrightarrow{h} \mathbb{R}_0^+ \cup \{\infty\}$, die die kürzeste Distanz zu einem Zielzustand schätzt.

Ein **Feature** ist eine beliebige Eigenschaft, die ein Zustand haben kann oder nicht.

Ein **Fakt** ist ein 2-Tupel $\langle v, x \rangle$ mit $v \in V$ und $x \in \mathbb{G}$ (V und $\mathbb{G}$ wie oben). Fakten sind somit einfache Features.

2.2 Pattern-Databases

Eine **Pattern-Database** ist ein Vereinfachung der Planungsaufgabe. Dabei werden bloss Variablen aus $\tilde{V} \subseteq V$ berücksichtigt. Die Menge $\tilde{V}$ wird dabei **Pattern** genannt.[1][2]

Ein **abstrakter Zustand** ist ein Repräsentant aller Zustände, die innerhalb einer Pattern-Database als äquivalent betrachtet werden. Mit anderen Worten sind das die Quasi-Zustände innerhalb einer Pattern-Database.

Pattern-Databases projizieren die Planungsaufgabe auf das Pattern. Dies bedeutet, dass eine Aktion als zulässig erachtet wird, wenn alle Pattern-Variablen die Bedingungen der Aktion erfüllen. Die abstrakten Zustände, die mindestens einen Zielzustand beinhalten, werden als Zielzustand betrachtet.

Es wird nun mit einer blinden Suche ein Plan für jeden abstrakten Zustand der Pattern-Database gefunden, sofern möglich. Ist es ausgeschlossen einen Plan zu finden, so wird Unendlich als h-Wert für den abstrakten Zustand gesetzt, ansonsten ist die Summe der Kosten aller Aktionen im Plan der h-Wert des abstrakten Zustandes.

Mindestens die Aktionen in so einem Plan müssen auch in einem Plan zur Lösung des eigentlichen Problems ausgeführt werden und somit ist so ein h-Wert eine Minimalschätzung von h^*, der exakten Distanz zum nächstgelegenen Zielzustand.

2.3 Potential-Heuristiken

Eine **Potential-Heuristik** ist eine Heuristik, die für eine Menge von Fakten jeweils einen Wert (**Potential**) besitzt, das zum h-Wert addiert wird, falls das Fakt wahr ist.[3]

Potential-Heuristiken lassen sich extrem schnell auswerten, da bloss ein Feature und je nach dem eine Addition für jedes Potential ausgewertet werden muss, um einen h-Wert zu berechnen. Selbst wenn die Features komplexer sind, geht das in der Regel mit deutlich weniger Rechenschritten, als mit anderen Heuristiken.

Andererseits benötigt eine Potential-Heuristik, dass man zuerst die Potentiale bestimmt, was unter Umständen grossen Rechenaufwand bedeuten kann. Manchmal ist es aber relativ unwichtig wie lange im Vorfeld gerechnet werden muss, sofern die Auswertungen danach effizient genug sind.

3

Algorithmen für die Berechnung von Potentialen einer Potential-Heuristik

3.1 Allgemeines

Die Zielsetzung dieser Arbeit bestand daraus Potentiale aufgrund von Pattern-Databases zu lernen, wofür man eine Potential-Heuristik benötigt. Also musste man festlegen was für Features verwendet werden sollten. So wurde beschlossen zu jedem Wert jeder Variable (alle Fakten) je ein Potential zuzuordnen.

Komplexere Features wären zwar möglich, jedoch würde das bedeuten, dass wohl sehr wesentlich mehr Potentiale hätten bestimmt werden müssen und es schien fraglich, ob sich das mit der eventuell verbesserten Performance in der Suche rechtfertigen liesse.

Ausserdem werden Pattern-Databases benötigt. Es gibt bereits Pattern-Collection-Generatoren und in dieser Arbeit war das Augenmerk auf die Verarbeitung der Informationen aus den Pattern-Databases gerichtet, also wurden zwei bereits existierende, vielversprechende Pattern-Collection-Generatoren eingesetzt, um die Pattern-Databases zu generieren.

Die Potentiale sollten nun so bestimmt werden, dass sie die Informationen aus den Pattern-Databases *möglichst genau* widerspiegeln. Wie üblich wurde die *Summe der Fehlerquadrate* als Mass der Ungenauigkeit gewählt. In diesem Fall ist dies die Summe der quadrierten Differenzen zwischen den h-Werten der Pattern-Database und den h-Werten der Potential-Heuristik.

Eine weitere Fragestellung war die Handhabe von Sackgassen (*Dead-Ends*). Wenn in einem Zustand kein Pfad mehr zum Zielzustand führen kann, so bleibt man dort stecken und kann nie mehr das Ziel erreichen. Für so einen Zustand ist $h^* = \infty$.

Zwar kann man nie Unendlich darstellen, aber man kann die grösstmögliche darstellbare Zahl stattdessen verwenden. Stellt man Potentiale so ein, dass sie dies reflektieren, so ergibt sich das Problem, dass die Unterschiede zwischen den h-Werten der Potential-Heuristik von

diesen grösstmöglichen Zahlen dominiert werden. Dies führt zwar schon dazu, dass die Suche weg von diesen Dead-Ends geführt wird, jedoch dominiert dann dieses Verhalten und die Suche wird quasi nur noch weg von Dead-Ends gelenkt und nicht mehr in Richtung eines Zielzustandes.

Es ist schwierig den Dead-Ends einen geeigneten h-Wert zuzuschreiben, so dass eine Balance zwischen dem Vermeiden von Dead-Ends und dem Suchen eines Zielzustandes in jedem Fall garantiert ist. Mit dieser Frage hat sich der Autor nicht tiefer auseinandergesetzt und in der Implementation wird den Dead-Ends einfach der Durchschnittliche h-Wert der Pattern-Database zugeschrieben, was einem Ignorieren der Dead-Ends gleich kommt.

3.2 Greedy Algorithmus

Eine der ersten Aufgaben war es, Algorithmen zu finden um die Potentiale zu lernen. Dabei stellte sich die Frage, wie ein Potential zu setzen wäre, wenn alle anderen Potentiale als gegeben angeschaut würden. Dies führte zum *Greedy-Algorithmus*.

3.2.1 Abweichungen eines Potentials

Betrachtet man ein Potential für sich alleine und erachtet alle anderen als gegeben, so ergibt sich als Abweichung des Potentials der Durchschnitt aller Fehler der Potential-Heuristik gegenüber den Pattern-Databases. Ist eine Reihe von Zahlen gegeben, so ist nämlich die Zahl, die ihre Fehlerquadrate zu der Reihe minimiert, gerade der Durchschnitt dieser Reihe von Zahlen.

Beweis:
Sei K ein Körper und $n \in \mathbb{N}$, seien $x_k \in K \ \forall k \in \mathbb{N}$ mit $k \leq n$.
Gesucht ist x, sodass

$$\sum_{k=1}^{n}(x - x_k)^2 \tag{3.1}$$

minimal wird.
Sei weiter $m = \frac{1}{n}\sum_{k=1}^{n} x_k$ der Durchschnitt der x_k. So gilt:
$\sum_{k=1}^{n}(x - x_k)^2 = \sum_{k=1}^{n}\left((x - m) - (x_k - m)\right)^2 =$
$= \sum_{k=1}^{n}\left((x - m)^2 - 2(x - m)(x_k - m) + (x_k - m)^2\right) =$
$= \left[\sum_{k=1}^{n}(x - m)^2\right] - \left[2(x - m)\sum_{k=1}^{n}(x_k - m)\right] + \left[\sum_{k=1}^{n}(x_k - m)^2\right] =$
$= n \cdot (x - m)^2 - 2(x - m)\left(\sum_{k=1}^{n}(x_k) - n \cdot m\right) + \sum_{k=1}^{n}(x_k - m)^2 =$
$= n \cdot (x - m)^2 - 2(x - m)(n \cdot m - n \cdot m) + \sum_{k=1}^{n}(x_k - m)^2 =$
$= n \cdot (x - m)^2 + \sum_{k=1}^{n}(x_k - m)^2$
$\sum_{k=1}^{n}(x_k - m)^2$ ist unabhängig von x und der Ausdruck wird minimal, wenn $n \cdot (x - m)^2$ minimal wird. Für $x = m$ wird das 0, da aber $n > 0$ und $(x - m)^2 \geq 0$ gilt, kann es nie negativ werden. Daraus folgt, dass $x = m$ die Fehlerquadrate 3.1 minimiert.

Dies erlaubt es ein Potential zu korrigieren, solange alle anderen als bereits gesetzt angeschaut werden. Folglich kann man mit einem beliebigen Satz von Potentialen anfan-

gen und eines nach dem anderen so korrigieren, dass die Fehlerquadrate gegenüber der Pattern-Database minimiert werden. So erhält man einen iterativen Algorithmus, der "greedy"getauft wurde, weil er "gierig", das heisst sobald als möglich, ein Potential, wenn auch ungenau, bestimmt.

Nachdem ein Potential verändert wird, müssen die Fehler zu den h-Werten der Pattern-Database neu berechnet, bzw. entsprechend angepasst werden. Alle Fehler jedes Mal neu zu berechnen würde wesentlich mehr Aufwand bedeuten, als eine Anpassung derjenigen Abweichungen, die vom neuen Potential beeinflusst werden, sofern diese Einflüsse bekannt sind.

Es muss also der Einfluss einer Änderung eines Potentials auf den Durchschnitt der h-Werte zu einem Fakt berechnet werden. Seien $v_i, v_j \in V$. Wird ein Potential von v_i um δ_i geändert, so bewirkt sie eine Änderung (δ_j) der durchschnittlichen Abweichung aller Potentiale von v_j. Sei n_j die Anzahl abstrakter Zustände in denen ein Fakt von v_j wahr ist; sei $n_{i,j}$ die Anzahl abstrakter Zustände in denen je ein Fakt von v_i und eines von v_j wahr ist. Es lässt sich leicht nachvollziehen, dass für δ_j gilt:

$$\delta_j = -\delta_i \cdot \frac{n_{i,j}}{n_j}$$

Um die Berechnung möglichst effizient zu gestalten, wurden die n_i in die Diagonale einer Matrix geschrieben und die $n_{i,j}$ in den Eintrag an der Stelle $[i, j]$.

In einer ersten Implementation wurde eine Liste von Fakten bzw.. Potentialen der Reihe nach abgearbeitet. Nach einem Durchlauf durch diese Liste änderte sich aber das Suchverhalten des Suchalgorithmus nicht mehr. Dies lag daran, dass alle Potentiale, die zu einer Variablen gehören, hintereinander abgearbeitet wurden, was zur Folge hatte, dass von da an immer nur alle diese Potentiale gleichermassen verändert wurden. Addiert man nämlich eine Konstante c in jedes Potential einer Variablen, so ändert sich der h-Wert für *alle* Zustände auch um c, weil immer genau ein Potential einer Variablen in den h-Wert addiert wird. Dies aber hat keinen Einfluss auf die Unterschiede in den h-Werten der Zustände. Es wurde so schliesslich klar, dass die Reihenfolge in der die Potentiale abgearbeitet werden, eine zentrale Rolle spielt.

Die Implementation wurde daraufhin so geändert, dass immer das Potential mit dem grössten Fehler zuerst abgearbeitet wird. Diese Methode bewies sich als sehr zuverlässig.

Procedure 1 Pseudocode des Greedy-Algorithmus

Input: PatternDatabaseCollection $PdbColl$, Variables V, VariableDomains D, InfluenceMatrix M
Output: PotentialsMatrix $potentials$

```
for all pot ∈ potentials do
    pot ← 0
end for
Matrix sums
Matrix nums
for all pdb ∈ PdbColl do
    for all var ∈ V do
        for all val ∈ D[var] do
            sums[var][val] ← sums[var][val] + pdb.GETSUMH(var, val)
            nums[var][val] ← nums[var][val] + pdb.GETNUMH(var, val)
        end for
    end for
end for
PriorityList deltaList
for all var ∈ V do
    for all val ∈ D[var] do
        delta ← sums[var][val]/nums[var][val]
        deltaList.PUSH(⟨var, val, delta⟩, |delta|)
    end for
end for
while |deltaList.PEAK.DELTA| > ε ∧ numIter < maxIter do
    delta ← deltaList.POP
    potentials[delta.var][delta.val] ← potentials[delta.var][delta.val] + delta.delta
    for all pdb ∈ PdbColl.WITHVAR(delta.var) do
        for all var2 ∈ pdb.PATTERN|var2 ≠ var do
            for all val2 ∈ D[var2] do
                correction ← delta.delta · M[delta.var][var2] / M[var2][var2]
                deltaList[var2][val2].delta ← deltaList[var2][val2] − correction
            end for
        end for
    end for
end while
```

3.3 Algebraischer Algorithmus

Eine andere Herangehensweise war die Betrachtung des Problems aus algebraischer Sicht, die ebenfalls zu einem Ergebnis führte.

3.3.1 Die Potentialfunktion

Ordnet man jedem Fakt den Wert 1 zu, falls es wahr ist, und sonst 0, so lässt sich jeder Zustand über einen Vektor charakterisieren, der diese Wahrheitswerte aller Fakten beinhaltet. Der Raum dieser Vektoren werde mit *Faktenraum* bezeichnet.

Die Potentialfunktion ordnet jedem Zustand im Zustandsraum einen h-Wert zu. Interpretiert man aber die Potentialfunktion als eine Abbildung aus dem Faktenraum auf die h-

Werte, so wird sie ein einfaches Standard-Skalarprodukt mit dem Potentialvektor (ein Vektor aus allen Potentialen). Dazu müssen selbstverständlich die Reihenfolge der Wahrheitswerte im Faktenvektor mit der Reihenfolge der Potentiale im Potentialvektor übereinstimmen. So erhält man folgenden Zusammenhang:

Sei $\vec{p}$ der Potentialvektor, $\vec{f}(s)$ der Faktenvektor zum Zustand s, sowie $h_{pot}(s)$ der h-Wert der Potential-Heuristik, so gilt:

$$\forall s \in S : h_{pot}(s) = \vec{p}^{\top} \cdot \vec{f}(s) \tag{3.2}$$

Wobei $\vec{a} \cdot \vec{b}$ das Standard-Skalarprodukt zwischen $\vec{a}$ und $\vec{b}$ und $\vec{a}^{\top}$ die Transponierte von $\vec{a}$ repräsentiert.

Schreibt man alle Faktenvektoren in die Zeilen einer Matrix A, sowie alle $h_{pot}(s)$ in einen Vektor $\vec{h}_{pot}$, so wird 3.2 zu:

$$\vec{h}_{pot} = A\vec{p} \tag{3.3}$$

Dies ist eine Matrix-Gleichung und zeigt somit, dass die Potentiale durch die Potentialfunktion über eine lineare Abbildung in den Zustandsraum überführt werden.

3.3.2 Kleinste Fehlerquadrate

Sei nun $\vec{h}_{soll}$ der Vektor der Werte, zu denen man die Fehlerquadrate minimieren will, so erhält man den Vektor der Fehler $\vec{e}$ über:

$$\vec{e} = \vec{h}_{pot} - \vec{h}_{soll} = A\vec{p} - \vec{h}_{soll} \tag{3.4}$$

Es soll nun also der Vektor $\vec{p}$ gefunden werden, der die Fehlerquadrate der Gleichung $A\vec{p} = \vec{h}_{soll}$ minimiert. Für so einen Vektor gilt:[4, p. 218]

$$A^{\top}A\vec{p} = A^{\top}\vec{h}_{soll} \tag{3.5}$$

3.3.3 Pattern-Databases

In dieser Arbeit sollen Pattern-Databases eingesetzt werden um die Potentiale zu lernen, somit müssen in $\vec{h}_{soll}$ die h-Werte der abstrakten Zustände geschrieben werden.

In einer Pattern-Database werden jeweils viele Zustände in einem abstrakten Zustand zusammengefasst. In diesen Zuständen haben jeweils die Pattern-Variablen gleiche Werte, aber sie unterscheiden sich in den anderen Variablen. Ein einzelner h-Wert einer Pattern-Database bezieht sich also auf viele verschiedene Zustände. $\vec{h}_{soll}$ beinhaltet folglich dieselben Werte in den Zeilen, die zu demselben abstrakten Zustand gehören.

Die h-Werte von Pattern-Databases beinhalten demnach keine Information über Potentiale, die nicht zu Pattern-Variablen gehören. Entsprechend kann man die Spalten von A,

sowie die zugehörigen Potentiale in $\vec{p}$, weglassen aus dem Gleichungssystem. Der so erhaltene, reduzierte Vektor werde mit $\vec{p}_{pdb}$ bezeichnet. Die Zeilen dieser Matrix, die zu demselben abstrakten Zustand gehören, sind nun auch jeweils identisch. Dies bedeutet, dass das Gleichungssystem identische Gleichungen beinhaltet, die demnach, bis auf je eine Gleichung, weggelassen werden können. Also sind entsprechende Zeilen der Matrix, sowie Einträge in $\vec{h}_{soll}$ wegzulassen. Dieser Vektor werde mit $\vec{h}_{pdb}$ bezeichnet und die Matrix mit A_{pdb}. So wird 3.5 zu:

$$A_{pdb}^{\top} A_{pdb} \vec{p}_{pdb} = A_{pdb}^{\top} \vec{h}_{pdb} \tag{3.6}$$

3.3.4 Die Matrix $A_{pdb}^{\top} A_{pdb}$

Die Einträge $a_{i,j}$ der Matrix $A_{pdb}^{\top} A_{pdb}$ berechnen sich als das Standard-Skalarprodukt des i-ten und j-ten Spaltenvektors von A_{pdb}. Die Spalten von A_{pdb} korrespondieren zu Fakten, ihre Zeilen zu abstrakten Zuständen und sie besitzt immer dort eine 1 wo das korrespondierende Fakt im korrespondierenden abstrakten Zustand wahr ist und sonst eine 0. In zwei Spalten der Matrix gibt es also immer dann an derselben Stelle eine 1, wenn die beiden korrespondierenden Fakten im korrespondierenden abstrakten Zustand wahr sind. Demnach ergibt das Skalarprodukt zweier Spaltenvektoren die Anzahl abstrakter Zustände, in denen beide Fakten wahr sind. Handelt es sich um zweimal denselben Spaltenvektor, so wird das natürlich die Anzahl abstrakter Zustände, in denen das zugehörige Fakt wahr ist.

Eine Pattern-Database besitzt einen abstrakten Zustand für jede Variablenbelegung der Pattern-Variablen. Seien $v_i \in V$, sei I wie folgt definiert: $I = \{i | v_i \in \tilde{V}\}$, sei d_i der Betrag der Domäne der Variable $v_i \in \tilde{V}$. Somit ist die Gesamtzahl abstrakter Zustände N_{as}:

$$N_{as} = \prod_{i \in I} d_i$$

Weil jedes Fakt einer Variablen in gleich vielen abstrakten Zuständen wahr ist, ist die Anzahl abstrakter Zustände n_i, in denen ein Fakt von v_i wahr ist:

$$n_i = \frac{N_{as}}{d_i}$$

Die Anzahl abstrakter Zustände, in denen zwei Fakten einer Variablen wahr sind, ist immer 0 (eine Variable hat nie zwei verschiedene Werte auf einmal).

Die Anzahl abstrakter Zustände $n_{i,j}$, in denen zwei Fakten von verschiedenen Variablen v_i, v_j mit $i \neq j$ wahr sind, ist:

$$n_{i,j} = \frac{n_i}{d_j} = \frac{N_{as}}{d_i d_j}$$

Ordnet man die Spalten von A_{pdb} so, dass immer die zu Fakten derselben Variablen zugehörigen Spalten nebeneinander zu liegen kommen, so erhält $A_{pdb}^{\top} A_{pdb}$ Blockgestalt. Mit $i, j \in I$ ergibt sich:

$$A_{pdb}^{\top} A_{pdb} = \left[B_{i,j} \right]$$

Ein Zeilenblock, bzw. ein Spaltenblock korrespondiert dabei zu je einer Pattern-Variablen und ihre Grösse (Anzahl Zeilen bzw. Spalten im Block) entsprechen dann den Beträgen der Domänen der zugehörigen Pattern-Variablen.

$$B_{i,j} \in \mathbb{N}_0^{d_i \times d_j}$$

Gilt $i \neq j$, so sind alle Einträge von $B_{i,j}$ gleich, da es immer gleich viele abstrakte Zustände gibt, in denen je ein Fakt der Variablen v_i und eines von v_j wahr ist. Somit gilt für $i \neq j$:

$$B_{i,j} = \begin{bmatrix} n_{i,j} \end{bmatrix}$$

Die Diagonal-Blöcke $B_{i,i}$ sind schliesslich $n_i \cdot E$, wobei E die Einheitsmatrix ist:

$$B_{i,i} = \begin{pmatrix} n_i & 0 & \dots & & 0 \\ 0 & n_i & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & n_i & 0 \\ 0 & \dots & & 0 & n_i \end{pmatrix}$$

Sei $\vec{b} = A_{pdb}^\top \vec{h}_{pdb}$. Der i-te Eintrag von $\vec{b}$ ist dann die Skalarmultiplikation der i-ten Spalte von A_{pdb} mit $\vec{h}_{pdb}$. Dies ist die Summe über die h-Werte der abstrakten Zustände in denen das zur i-ten Spalte korrespondierende Fakt wahr ist. Sei $\tilde{s}_i$ der zur i-ten Zeile von A_{pdb} korrespondierende abstrakte Zustand, sei $\vec{f}_{pdb}(\tilde{s}_i)$ der auf die Fakten der Pattern-Database reduzierte Faktenvektor zu $\tilde{s}_i$ und sei $f_k(\tilde{s}_i)$ sein k-ter Eintrag. Sei $J_i = \{j | f_i(\tilde{s}_j) = 1\}$ und sei schliesslich $h_{pdb,j}$ der j-te Eintrag von $\vec{h}_{pdb}$. So gilt:

$$b_i = \sum_{j \in J_i} h_{pdb,j} \tag{3.7}$$

3.3.5 Die Lösung von $A_{pdb}^\top A_{pdb} \vec{p}_{pdb} = A_{pdb}^\top \vec{h}_{pdb}$

Das Gleichungssystem 3.6 lässt sich zwar so wie es ist mit dem Gauss-Algorithmus lösen, es kann aber mit relativ geringem Aufwand in mehrere kleinere Gleichungssysteme separiert werden.

Addiert man zur letzten Zeile eines Zeilenblocks von $A_{pdb}^\top A_{pdb}$ alle anderen Zeilen des Zeilenblocks, so wird diese letzte Zeile in allen Zeilenblöcken zu:

$$(\underbrace{n_i \dots n_i}_{d_i\text{-mal}} \dots \underbrace{n_j \dots n_j}_{d_j\text{-mal}} \dots) \tag{3.8}$$

Multipliziert man nun alle Zeilen des Zeilenblocks zur Variablen v_i ausser dieser letzten Zeile mit d_i, so werden alle Zeilen in allen Nicht-Diagonal-Blöcken identisch zum entsprechenden Teil der letzten Zeile jedes Zeilenblocks.

Nun muss man nur noch die letzte Zeile eines beliebigen Zeilenblocks (sie sind je alle identisch), von allen anderen Zeilen subtrahieren. Angenommen man wählt den letzten Zeilenblock, so werden alle Nicht-Diagonal-Blöcke ausschliesslich aus Nullen bestehen, ausser in der letzten Zeile der (ganzen) Matrix, wo immer noch 3.8 steht. Die letzten Zeilen jedes anderen Zeilenblocks sind jetzt (auch im Diagonal-Block) Nullzeilen. Sie repräsentieren Freiheitsgrade, die genutzt werden können um das zugehörige Potential auf 0 zu setzen, was das Streichen der zugehörigen Spalten ermöglicht.

So werden alle Diagonal-Blöcke, ausser dem Letzten, aus dem Gleichungssystem separiert und können einzeln gelöst werden. Diese Diagonal-Blöcke ($\tilde{B}_{i,i}$) haben nun folgende Gestalt:

$$\tilde{B}_{i,i} = N_{as} \cdot E - \left[n_i\right] = \begin{pmatrix} (N_{as}-n_i) & -n_i & \dots & & -n_i \\ -n_i & (N_{as}-n_i) & -n_i & \dots & -n_i \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -n_i & \dots & -n_i & (N_{as}-n_i) & -n_i \\ -n_i & \dots & & -n_i & (N_{as}-n_i) \end{pmatrix} \tag{3.9}$$

Bemerkung: $\tilde{B}_{i,i}$ hat eine Zeile und eine Spalte weniger als $B_{i,i}$.

Diese Operationen müssen selbstverständlich auch mit dem Vektor $\vec{b}$ durchgeführt werden. Nach 3.7 steht in b_i die Summe aller h-Werte von abstrakten Zuständen, in denen das zu b_i gehörende Fakt wahr ist. Summiert man im ersten Schritt alle b_i die zu Fakten einer Variablen gehören, so wird das die Summe über alle h-Werte von abstrakten Zuständen, in denen irgendein zu einer Variablen gehörendes Fakt wahr ist. Da dies in allen (abstrakten) Zuständen wahr ist, ist das die Summe über alle h-Werte der Pattern-Database. Dies ist selbstverständlich immer gleich viel, egal um welche Variable es sich handelt. Nun multipliziert man jedes andere b_i mit der Domänengrösse seiner zugehörigen Variablen und subtrahiert schliesslich die Summe aller h-Werte. An der Stelle der jeweils letzten b_i, die zu einer Variablen gehören, entstehen so ebenfalls Nullen, also waren die gestrichenen Nullzeilen der Matrix tatsächlich Nullzeilen des ganzen Gleichungssystems. Sei $v_{k(i)}$ die zu b_i gehörige Variable, sei S_h die Summe aller h-Werte der Pattern-Database und sei $\tilde{b}_i$ der i-te Eintrag des Ergebnisses, so ist für alle $\tilde{b}_i$ ausser dem letzten zu jeder Variablen:

$$\tilde{b}_i = d_{k(i)} \cdot b_i - S_h \tag{3.10}$$

Die letzten $\tilde{b}_i$ zu jeder ausser der letzten Variablen sind 0 und da ihre Zeilen aus der Matrix gestrichen wurden, sind auch sie zu streichen. Das letzte $\tilde{b}_n$ ist schliesslich S_h, wobei n die Anzahl Fakten ist.

Nachdem man nun die separierten Systeme gelöst hat, darf man nicht vergessen, dass da noch dieser letzte, nicht separierte, Zeilenblock übrig war. Aus dem letzten $\tilde{b}_n$ (das ja S_h ist) muss man dann jedes berechnete Potential, dessen Variable v_i sei, mit n_i multipliziert subtrahieren, bevor man den letzten Zeilenblock auch noch lösen kann.

So erhält man alle Vorschläge einer Pattern-Database für alle Potentiale aller Pattern-Variablen. In der Implementation wurde dann der Durchschnitt über die Vorschläge aus allen Pattern-Databases als Potential gewählt.

3.3.6 Beispiel

Sei folgende Pattern-Database gegeben:

$v_0 \backslash v_1$	0	1	2
0	1	2	3
1	0	2	1

$$A_pdb = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \vec{h}_{pdb} = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 2 \\ 3 \\ 1 \end{pmatrix}$$

$$A_{pdb}^{\top} A_{pdb} = \begin{pmatrix} 3 & 0 & 1 & 1 & 1 \\ 0 & 3 & 1 & 1 & 1 \\ 1 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 0 & 2 \end{pmatrix}$$

$$A_{pdb}^{\top} \vec{h}_{pdb} = \begin{pmatrix} 1+2+3 \\ 0+2+1 \\ 1+0 \\ 2+2 \\ 3+1 \end{pmatrix} = \begin{pmatrix} 6 \\ 3 \\ 1 \\ 4 \\ 4 \end{pmatrix}$$

Nun werden in $A_{pdb}^{\top} A_{pdb}$ die Zeilen zur Letzten des Zeilenblocks addiert und dann werden alle Zeilen ausser der letzten jedes Zeilenblocks mit der Domänengrösse ihrer zugehörigen Variablen multipliziert:

$$A_{pdb}^{\top} A_{pdb} = \begin{pmatrix} 3 & 0 & 1 & 1 & 1 \\ 0 & 3 & 1 & 1 & 1 \\ 1 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 0 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 0 & 1 & 1 & 1 \\ 3 & 3 & 2 & 2 & 2 \\ 1 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 2 & 0 \\ 3 & 3 & 2 & 2 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} 6 & 0 & 2 & 2 & 2 \\ 3 & 3 & 2 & 2 & 2 \\ 3 & 3 & 6 & 0 & 0 \\ 3 & 3 & 0 & 6 & 0 \\ 3 & 3 & 2 & 2 & 2 \end{pmatrix}$$

Nun subtrahiert man die letzte Zeile aus allen anderen Zeilen:

$$\rightarrow \begin{pmatrix} 3 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & -2 & -2 \\ 0 & 0 & -2 & 4 & -2 \\ 3 & 3 & 2 & 2 & 2 \end{pmatrix}$$

Dasselbe für $A_{pdb}^{\top}\vec{h}_{pdb}$:

$$\begin{pmatrix} 6 \\ 3 \\ 1 \\ 4 \\ 4 \end{pmatrix} \rightarrow \begin{pmatrix} 6 \\ 9 \\ 1 \\ 4 \\ 9 \end{pmatrix} \rightarrow \begin{pmatrix} 12 \\ 9 \\ 3 \\ 12 \\ 9 \end{pmatrix} \rightarrow \begin{pmatrix} 3 \\ 0 \\ -6 \\ 3 \\ 9 \end{pmatrix}$$

Die Nullzeile wird jetzt gestrichen, das Potential von $v_0 = 1$ wird auf 0 gesetzt und dann auch die Spalte gestrichen:

$$\begin{pmatrix} 3 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & -2 & -2 \\ 0 & 0 & -2 & 4 & -2 \\ 3 & 3 & 2 & 2 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & -3 & 0 & 0 & 0 \\ 0 & 0 & 4 & -2 & -2 \\ 0 & 0 & -2 & 4 & -2 \\ 3 & 3 & 2 & 2 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 4 & -2 & -2 \\ 0 & -2 & 4 & -2 \\ 3 & 2 & 2 & 2 \end{pmatrix}$$

Der Block zur Variablen v_0 ist jetzt separiert (er besteht jetzt nur noch aus einem einzigen Eintrag) und kann gelöst werden:
$3p_0 = 3 \implies p_0 = 1$
Das muss jetzt aus dem letzten Eintrag in $\vec{b}$ eliminiert werden:

$$\rightarrow \begin{pmatrix} 3 \\ -6 \\ 3 \\ 9 \end{pmatrix} \rightarrow \begin{pmatrix} 3 \\ -6 \\ 3 \\ 9 - 3 \cdot 1 \end{pmatrix} = \begin{pmatrix} 3 \\ -6 \\ 3 \\ 6 \end{pmatrix}$$

Schliesslich löst man den letzten Block:

$$\left(\begin{array}{ccc|c} 4 & -2 & -2 & -6 \\ -2 & 4 & -2 & 3 \\ 2 & 2 & 2 & 6 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & -2 & -2 & -6 \\ 0 & 3 & -3 & 0 \\ 0 & 3 & 3 & 9 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & 0 & -4 & -4 \\ 0 & 3 & -3 & 0 \\ 0 & 0 & 6 & 9 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & 0 & 0 & 2 \\ 0 & 3 & 0 & \frac{9}{2} \\ 0 & 0 & 1 & \frac{3}{2} \end{array}\right) \rightarrow$$
$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \frac{3}{2} \\ 0 & 0 & 1 & \frac{3}{2} \end{array}\right)$$

Also haben wir folgenden $\vec{p}$ erhalten: $\begin{pmatrix} 1 \\ 0 \\ \frac{1}{2} \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix}$

Procedure 2 Pseudocode des algebraischen Algorithmus

Input: PatternDatabaseCollection $PdbColl$, VariableDomains D, BiiMatricesMatrix $Biis$
Output: PotentialsMatrix $potentials$

```
for all pot ∈ potentials do
    pot ← 0
end for
for all pdb ∈ PdbColl do
    lastB ← pdb.getTotalSum
    N_as ← pdb.getTotalNum
    for all var ∈ pdb.pattern except last do
        for all val ∈ D[var] except last do
            Vector b̃
            b̃[val] ← D[var] · pdb.getSumH(var, val) − pdb.getTotalSum
        end for
        Vector potSuggest ← solveGauss(Biis[pdb][var], b̃)
        for all val ∈ D[var] except last do
            potentials[var][val] ← potentials[var][val] + potSuggest[val] / pdbColl.getNumPatternsWith(var)
            lastB ← lastB − potSuggest·N_as / |D[var]|
        end for
    end for
    var ← pdb.pattern.last
    for all val ∈ D[var] except last do
        Vector b̃
        b̃[val] ← |D[var]| · pdb.getSumH(var, val) − pdb.getTotalSum
    end for
    b̃.append(lastB)
    Vector potSuggest ← solveGauss(Biis[pdb][var], b̃)
    for all val ∈ D[var] do
        potentials[var][val] ← potentials[var][val] + potSuggest[val] / pdbColl.getNumPatternsWith(var)
    end for
end for
```

4

Experimente

Die Implementation der vorgestellten Algorithmen erfolgte im Fast-Downward Planning System[5] und die Experimente wurden mittels lab[6] auf dem Maia-Cluster der Universität Basel durchgeführt. Als Suite wurde stets dieselbe Auswahl von Satisficing-Problemen aus den Fast-Downward Benchmarks gewählt. Diese bestehen weitgehend aus Benchmarks der International Planning Competition (IPC).

4.1 Anzahl Iterationen des Greedy-Algorithmus

Der Greedy-Algorithmus korrigiert in jeder Iteration dasjenige Potential, welches die grösste Korrektur benötigt. Mit n Iterationen werden also höchstens n Potentiale $p_i \neq 0$. Es wurde angenommen, dass mit mehr Iterationen die Qualität der Heuristik zunehmen würde und dass es mindestens so viele Iterationen braucht für eine sinnvolle Heuristik, wie es Fakten im Problem gibt.

In einem ersten Experiment wurde also versucht festzustellen in welchen Problemstellungen wie viel Iterationen benötigt werden, um eine gute Heuristik zu erhalten. Erstaunlicherweise ergab dieses Experiment aber, dass in einigen Fällen bereits nach wenigen Iterationen eine starke Heuristik resultiert, die dann aber mit zunehmender Anzahl Iterationen schwächer wird.

Besonders deutlich zu sehen war dieser Effekt in den Planungsdomänen “gripper” und “miconic”. In den Domänen “pegsol-08-strips” und “pegsol-sat11-strips” war dieses Verhalten in der Tabelle der Expansionen ebenfalls ersichtlich.

In der Tabelle der Coverage (Tabelle 4.1) lässt sich dies zwar weniger deutlich sehen als in den Tabellen der Expansionen in einzelnen Planungsdomänen. Da diese aber relativ gross sind, soll die Tabelle der Expansionen in “gripper” exemplarisch dienen. Das Experiment wurde ausser auf den in Tabelle 4.1 aufgeführten Problemdomänen auch auf weiteren durchgeführt, jedoch konnte keine der Konfigurationen auch nur eines dieser Probleme lösen. Diese Domänen waren: barman-sat11-strips, barman-sat14-strips, childsnack-sat14-strips,

elevators-sat11-strips, floortile-sat11-strips, floortile-sat14-strips, ged-sat14-strips, openstacks-sat11-strips, openstacks-sat14-strips, parking-sat11-strips, parking-sat14-strips, tetris-sat14-strips, transport-sat14-strips und visitall-sat14-strips.

In Tabelle 4.1 ist hinter der Bezeichnung der Problemklassen in Klammern die gesamte Anzahl Probleme der Klasse angegeben. "miconic" ist mit 150 Problemen hier eindeutig die grösste Klasse und hat somit die grösste Bedeutung für die totale Coverage. Wie aber bereits erwähnt, nimmt in dieser Klasse die Qualität der Heuristik ab einem gewissen Punkt tendenziell ab mit zunehmender Anzahl Iterationen. Somit ist die Coverage der Konfiguration mit 50'000 Iterationen nur deswegen nicht die Beste in der Gesamt-Coverage.

4.2 Steuerung der Iteration über die Fehlertoleranz ε

Der Greedy-Algorithmus kann auch anstatt über die maximale Anzahl Iterationen auch über die Wahl von ε gesteuert werden. Die Iteration bricht nämlich ab, wenn entweder die maximale Anzahl Iterationen erreicht wird oder wenn der grösste Fehler in den Potentialen ein ε unterschreitet.

Weil es offensichtlich manchmal Sinn macht frühzeitig die Iteration abzubrechen, stellt sich die Frage, ob es mehr Sinn macht dies über das ε oder die maximale Anzahl Iterationen zu machen. Im nächsten Experiment wurde also das Verhalten mit verschiedenen ε untersucht.

Es ist zwar im Prinzip dasselbe, was zu sehen ist in Tabelle 4.2 und Tabelle 4.3, aber in der Letzteren ist die Qualität der Konfigurationen konsistenter durch die einzelnen Probleme. Es scheint darum mehr Sinn zu machen über das ε zu steuern.

4.3 Pattern-Generatoren

In Fast-Downward sind einige Pattern-Generatoren implementiert. Als mögliche geeignete Kandidaten für die Verwendung in dieser Arbeit zeigten sich insbesondere der "systematic pattern generator"[7] und der "hillclimbing pattern generator"[8]. Ein Experiment wurde durchgeführt um die beiden bezüglich ihrer Eignung zu testen. Dabei wurde der Greedy-Algorithmus mit $\varepsilon = 0.01$, sowie der algebraische Algorithmus eingesetzt.

4.4 Pattern-Filter

Pattern-Generatoren generieren in der Regel Patterns unterschiedlicher Grösse. Somit haben die Pattern-Databases unter Umständen stark unterschiedliche Anzahl von abstrakten Zuständen, was ihre Vergleichbarkeit einschränkt. Um dies zu vermeiden wurde ein Filter eingesetzt um Patterns gleicher Länge zu erhalten.

Coverage									
Iterationen	200	500	1000	2000	5000	10000	20000	50000	algebraisch
airport (50)	19	18	18	18	19	18	18	18	18
blocks (35)	35	35	35	35	35	35	35	35	35
depot (22)	13	12	12	12	12	12	12	12	12
driverlog (20)	13	15	15	16	16	16	16	16	18
elevators-sat08-strips (30)	4	4	4	4	4	4	4	4	4
freecell (80)	74	77	75	75	76	76	76	76	76
grid (5)	2	1	1	1	1	1	1	1	1
gripper (20)	**20**	**16**	**13**	13	13	13	13	13	9
hiking-sat14-strips (20)	19	19	19	18	18	19	18	18	2
logistics00 (28)	24	28	28	28	28	28	28	28	28
logistics98 (35)	8	13	18	21	21	21	21	21	11
miconic (150)	150	150	150	**150**	**149**	**114**	**110**	**109**	71
movie (30)	30	29	30	30	30	30	30	30	30
mprime (35)	24	23	23	23	23	23	23	23	21
mystery (30)	17	16	17	17	17	17	17	16	16
nomystery-sat11-strips (20)	12	10	10	9	8	9	8	9	9
openstacks-sat08-strips (30)	6	6	6	6	6	6	6	6	6
openstacks-strips (30)	7	7	7	7	7	7	7	7	11
parcprinter-08-strips (30)	16	16	15	15	16	16	16	16	20
parcprinter-sat11-strips (20)	4	4	3	4	4	4	4	4	5
pathways-noneg (30)	4	4	4	4	4	4	4	4	5
pegsol-08-strips (30)	30	30	30	30	30	30	30	30	30
pegsol-sat11-strips (20)	20	20	20	20	20	20	20	20	20
pipesworld-notankage (50)	24	24	24	24	24	25	27	27	25
pipesworld-tankage (50)	15	17	18	16	16	16	16	16	15
psr-small (50)	50	50	49	49	49	50	50	50	49
rovers (40)	5	5	5	5	5	5	5	5	10
satellite (36)	14	14	14	15	14	14	14	14	6
scanalyzer-08-strips (30)	28	30	30	30	30	30	30	30	26
scanalyzer-sat11-strips (20)	18	20	20	20	20	20	20	20	16
sokoban-sat08-strips (30)	17	16	16	16	16	16	17	17	17
sokoban-sat11-strips (20)	8	7	7	7	7	7	8	8	8
storage (30)	18	17	17	17	17	17	17	17	17
thoughtful-sat14-strips (20)	5	5	5	5	5	5	5	5	5
tidybot-sat11-strips (20)	3	3	3	3	3	3	3	3	14
tpp (30)	6	6	6	6	6	6	6	6	10
transport-sat08-strips (30)	16	13	15	18	21	25	24	25	13
transport-sat11-strips (20)	1	0	0	1	2	5	4	5	0
trucks-strips (30)	8	8	8	8	8	8	8	8	7
visitall-sat11-strips (20)	2	5	5	5	5	5	5	5	4
woodworking-sat08-strips (30)	14	16	16	16	16	16	16	16	6
woodworking-sat11-strips (20)	1	2	2	2	2	2	2	2	1
zenotravel (20)	12	14	15	15	15	15	16	16	14
Summe	816	825	828	834	**839**	**813**	**810**	**811**	721

Tabelle 4.1: Coverage: Vergleich des Greedy-Algorithmus nach Anzahl Iterationen und zum algebraischen Algorithmus

Expansionen - gripper								
Iterationen	200	500	1000	2000	5000	10000	20000	50000
prob01	34	34	34	34	34	34	34	34
prob02	54	47	47	47	47	47	47	47
prob03	87	87	76	76	76	76	76	76
prob04	120	102	102	102	102	102	102	102
prob05	139	201	224	224	224	224	224	224
prob06	226	454	468	468	468	468	468	468
prob07	323	2444	2428	2428	2428	2428	2428	2428
prob08	672	5088	5129	5129	5129	5129	5129	5129
prob09	550	25176	42309	42287	42287	42287	42287	42287
prob10	916	57400	274290	274304	274304	274304	274304	274304
prob11	2127	161549	693820	693826	693826	693826	693826	693826
prob12	5708	504909	2777991	2777933	2777933	2777933	2777933	2777933
prob13	11496	1182662	6385132	6385174	6385174	6385174	6385174	6385174
prob14	15285	1771854	-	-	-	-	-	-
prob15	10086	6245373	-	-	-	-	-	-
prob16	1531	14693135	-	-	-	-	-	-
prob17	1032	-	-	-	-	-	-	-
prob18	1081	-	-	-	-	-	-	-
prob19	1097	-	-	-	-	-	-	-
prob20	1567	-	-	-	-	-	-	-

Tabelle 4.2: Expansionen der Problemklasse gripper: Vergleich nach Anzahl Iterationen

Expansionen - gripper						
ε	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
prob01	253	28	30	34	34	34
prob02	1853	45	41	54	47	47
prob03	11773	71	73	87	87	76
prob04	68605	125	106	102	102	102
prob05	376829	166	115	201	224	224
prob06	1982461	185	178	468	446	468
prob07	10092541	245	201	2444	2428	2428
prob08	-	285	363	5090	5129	5129
prob09	-	348	420	25199	42309	42287
prob10	-	412	916	57470	274276	274304
prob11	-	**478**	4940	625197	693826	693826
prob12	-	**654**	10620	2736504	2777947	2777933
prob13	-	**690**	41601	6385123	6385178	6385174
prob14	-	**836**	98838	-	-	-
prob15	-	**990**	158002	-	-	-
prob16	-	**1090**	1034228	-	-	-
prob17	-	1148	2060970	-	-	-
prob18	-	1191	3296454	-	-	-
prob19	-	1226	-	-	-	-
prob20	-	1503	-	-	-	-

Tabelle 4.3: Expansionen der Problemklasse gripper: Vergleich nach ε

4.5 Vergleich mit der FF-Heuristik

Wie nahe die in dieser Arbeit erstellten Heuristiken an eine der stärksten, bekannten Heuristiken kommt, wurde schliesslich auch noch angeschaut.

Coverage				
Pattern-Generator	Systematic		Hillclimbing	
Algorithmus	Greedy	Algebraic	Greedy	Algebraic
Summe	839	721	476	579

Tabelle 4.4: Summe der Coverage: Vergleich von Pattern-Generatoren

Coverage				
Konfiguration	ohne Filter		mit Filter	
Algorithmus	Greedy	Algebraic	Greedy	Algebraic
Summe	839	**721**	**908**	706

Tabelle 4.5: Summe der Coverage: Vergleich ohne und mit Pattern-Filter

Dieses Experiment zeigt klar, dass die FF-Heuristik etwas stärker ist als die im Rahmen dieser Arbeit entwickelten Heuristiken. Es gibt aber immerhin einige Problemklassen in denen der Greedy-Algorithmus die stärkste Heuristik liefert und mit "driverlog" gab es auch noch eine Klasse in der der algebraische Algorithmus das beste Resultat lieferte.

Schliesslich wurden noch die besten Konfigurationen der hier erarbeiteten Heuristiken mit der FF-Heuristik verglichen (siehe Tabelle 4.6). Diese Konfigurationen waren:

- Greedy-Algorithmus: 5'000 Iterationen und ein systematic pattern generator mit Pattern-Filter
- Algebraischer Algorithmus: systematic pattern generator ohne Pattern-Filter

4.6 Verschlechterung des Greedy-Algorithmus mit zunehmender Anzahl Iterationen

Die Frage weshalb mit mehr Iterationen der Greedy-Algorithmus sich nicht mehr verbessert, sondern sogar verschlechtert, ist eine der herausstechendsten Fragen die die Experimente aufgeworfen haben. Man kann spekulieren, dass eine Art "overfitting" geschieht. Overfitting geschieht in der Regel aber erst wenn die Anzahl Parameter und die Anzahl Datenpunkte ähnlich gross werden. Angesichts dessen, dass in den meisten Fällen aber Hunderte Potentiale auf Zehntausende Datenpunkte gefittet wurden, ist overfitting wohl eher auszuschliessen.

Mehr Sinn scheint die Spekulation zu machen, dass sehr unterschiedlich "gute" Pattern-Databases generiert wurden. Die Pattern-Databases, die grosse h-Werte liefern, sollten tendenziell besser sein, da sie ja eine Minimalschätzung von h^* machen.

Einzelne Tests wurden durchgeführt in denen die Pattern-Databases verworfen wurden, die die kleinsten durchschnittlichen h-Werte lieferten. Dies hatte aber einen klar negativen Effekt auf die Heuristik und so musste auch dieser Erklärungsversuch vorerst verworfen werden. Es könnte aber sein, dass gewisse Pattern-Databases mit kleinen h-Werten durchaus essentiell sind, aber eine Mehrheit eher störend wirkt.

Coverage			
Konfiguration	Greedy-best	Algebraic-best	FF
airport	19	18	**31**
barman-sat11-strips	3	0	**4**
barman-sat14-strips	0	0	**5**
blocks	35	35	35
childsnack-sat14-strips	0	0	**1**
depot	12	12	**16**
driverlog	16	**18**	17
elevators-sat08-strips	4	4	**11**
floortile-sat11-strips	0	0	**8**
floortile-sat14-strips	0	0	**2**
freecell	76	76	**79**
ged-sat14-strips	0	0	0
grid	1	1	**4**
gripper	13	9	**20**
hiking-sat14-strips	18	2	**20**
logistics00	28	28	28
logistics98	21	11	**29**
miconic	149	71	**150**
movie	30	30	30
mprime	23	21	**32**
mystery	17	16	17
nomystery-sat11-strips	8	9	9
openstacks-sat08-strips	6	6	6
openstacks-strips	24	11	**28**
parcprinter-08-strips	16	20	**22**
parcprinter-sat11-strips	4	5	5
parking-sat11-strips	0	0	**20**
parking-sat14-strips	0	0	**11**
pathways-noneg	5	5	**9**
pegsol-08-strips	30	30	30
pegsol-sat11-strips	20	20	20
pipesworld-notankage	24	25	**30**
pipesworld-tankage	16	15	**23**
psr-small	50	49	50
rovers	19	10	**26**
satellite	16	6	**28**
scanalyzer-08-strips	**30**	26	28
scanalyzer-sat11-strips	**20**	16	18
sokoban-sat08-strips	16	17	**28**
sokoban-sat11-strips	7	8	**18**
storage	17	17	**19**
tetris-sat14-strips	1	0	**9**
thoughtful-sat14-strips	5	5	**8**
tidybot-sat11-strips	**17**	14	16
tpp	23	10	23
transport-sat08-strips	**21**	13	13
transport-sat11-strips	**2**	0	0
trucks-strips	8	7	**14**
visitall-sat11-strips	**5**	4	3
woodworking-sat08-strips	16	6	**28**
woodworking-sat11-strips	2	1	**13**
zenotravel	15	14	**20**
Summe	908	721	**1115**

Tabelle 4.6: Coverage: Vergleich von FF zu Greedy mit 5'000 Iterationen und systematic pattern generator mit Pattern-Filter sowie zum algebraischen Algorithmus mit systematic pattern generator ohne Pattern-Filter

5
Schlussfolgerungen

5.1 Folgerungen aus den Experimenten

5.1.1 Vergleich zwischen algebraischem und Greedy-Algorithmus

Der algebraische hat etwas enttäuschend abgeschnitten gegenüber dem Greedy-Algorithmus. Der wesentlichste Unterschied sollte sein, dass der algebraische Algorithmus alle Pattern-Databases als gleichwertig anschaut (es wird der Durchschnitt gebildet aus den Vorschlägen aller Pattern-Databases), der Greedy-Algorithmus hingegen versucht die Fehlerquadrate zu den einzelnen abstrakten Zuständen zu minimieren. Somit erhalten Pattern-Databases mit vielen abstrakten Zuständen deutlich mehr Gewicht.

Ob dies nun bedeutet, dass man grossen Pattern-Databases mehr Gewicht zuordnen sollte bleibt aber etwas unklar, denn etwas in dieser Richtung wurde ja versucht mit dem Weglassen der Pattern-Databases mit den tiefsten durchschnittlichen h-Werten, was aber eher nicht zu besserer Performance führte. Somit bleibt eine weitere Frage offen.

5.1.2 Vergleich der Pattern-Generatoren

Der Hillclimbing-Generator hat ebenfalls nicht überzeugen können im Vergleich mit dem Systematic-Generator. Der Greedy-Algorithmus benötigte vergleichsweise wenige Iterationen, um die Potentiale soweit zu bestimmen, dass weitere Iterationen kaum mehr Sinn machten. Dies könnte vielleicht daran liegen, dass schlicht nicht genügend viele Patterns generiert wurden.

Das Dominance-Pruning des Hillclimbing-Generators könnte allenfalls auch eher schädlich sein, da mehrfache Informationen in beiden Ansätzen dazu führen, dass dieser Information mehr Gewicht zukommt. Filtert man aber dominierte Information aus den Patterns heraus, schwächt man somit das Gewicht dieser Information.

5.2 Weiterführende Arbeit

Der Autor ist der Meinung, dass sich durchaus einiges Potential zeigt in diesen Heuristiken, aber es wurden viele neue offene Fragen gefunden.

Ein einfaches Minimieren der Fehlerquadrate ist sehr wahrscheinlich suboptimal. Dies zeigt sich auch schon an der teils sehr unterschiedlichen Performance des Greedy- und des algebraischen Algorithmus. In den meisten Problemklassen scheint es mehr Sinn zu machen die abstrakten Zustände gleich zu gewichten, jedoch ist in ein paar wenigen Problemklassen der algebraische Algorithmus stärker. Dies weist darauf hin, dass dort die Pattern-Databases ähnlich stark zu gewichten wären. Vielleicht könnte man einen Weg finden aus statistischen Analysen der h-Werte herauslesen, wie stark welche Fehler zu gewichten sind.

Wie schon erwähnt sind gewisse Dinge (Dominance-Pruning) die für Pattern-Generatoren in den meisten Fällen sinnvoll sind in diesem Fall eher weniger sinnvoll. Ein spezifisch für diese Sache konzipierter Pattern-Generator könnte unter Umständen eine enorme Verbesserung bewirken.

Um weitere Forschung an Potential-Heuristiken zu vereinfachen könnte auch ein allgemeiner Potential-Rechner für fast-downward entwickelt werden, der aufgrund von gegebenen Samples die Potentiale bestimmt. Somit könnte man beliebige Methoden einsetzen um an h-Werte zu kommen um die Potentiale zu bestimmen. Möglicherweise reicht schon eine relativ kleine Stichprobe einer starken Heuristik, die vielleicht sehr aufwändig zu berechnen ist, um eine vergleichsweise effiziente Potential-Heuristik zu erhalten.

Literaturverzeichnis

[1] Culberson, J. C. and Schaeffer, J. Pattern-Databases. *Computational Intelligence*, 14(3):318–334 (1998).

[2] Edelkamp, S. Planning with Pattern Databases. In Cesta, A. and Borrajo, D., editors, *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 84–90. AAAI Press (2001).

[3] Pommerening, F., Helmert, M., Röger, G., and Seipp, J. From Non-Negative to General Operator Cost Partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 3335–3341. AAAI Press (2015).

[4] Strang, G. *Introduction to Linear Algebra*. Wellesley-Cambridge Press and SIAM, fifth edition (2016).

[5] Helmert, M. The Fast Downward Planning System. *Journal of Artifcial Intelligence Research*, 26:191–246 (2009).

[6] Seipp, J. lab. Https://bitbucket.org/jendrikseipp/lab.

[7] Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 1007–1012. AAAI Press (2007).

[8] Pommerening, F., Röger, G., and Helmert, M. Getting the Most Out of Pattern Databases for Classical Planning. In Rossi, F., editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 2357–2364 (2013).

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Andreas Ferenczi

Matriculation number — Matrikelnummer
2002-921-013

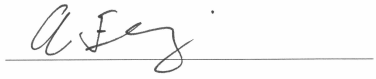
Title of work — Titel der Arbeit
Berechnung von Potential-Heuristiken aufgrund von Pattern-Databases

Type of work — Typ der Arbeit
Bachelorarbeit

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 21. Dezember 2016



Signature — Unterschrift