UNIVERSITÄT BASEL

Kontext-basierte Suche für klassische Handlungsplanung

Bachelorarbeit

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

Prüfer: Prof. Dr. Malte Helmert
Betreuer: Dr. Martin Wehrle, Manuel Heusner

Lukas Songajlo
lukas.songajlo@stud.unibas.ch

2. August 2014

Danksagung

An dieser Stelle möchte ich mich besonders bei Dr. Martin Wehrle und Manuel Heusner für die grossartige und intensive Betreuung bedanken. Ich fühlte mich jederzeit bestens aufgehoben. Ebenso geht ein großer Dank an Prof. Dr. Malte Helmert. Trotz anfänglich fehlender Vorkenntnis bekam ich die Möglichkeit im Fachbereich der künstlichen Intelligenz eine Abschlussarbeit zu schreiben.

Zusammenfassung

Ziel klassischer Handlungsplanung ist es auf eine möglichst effiziente Weise gegebene Planungsprobleme zu lösen. Die Lösung bzw. der *Plan* eines Planungsproblems ist eine Sequenz von Operatoren mit denen man von einem Anfangszustand in einen Zielzustand gelangt. Um einen Zielzustand gezielter zu finden, verwenden einige Suchalgorithmen eine zusätzliche Information über den Zustandsraum - die Heuristik. Sie schätzt, ausgehend von einem Zustand den Abstand zum Zielzustand. Demnach wäre es ideal, wenn jeder neue besuchte Zustand einen kleineren heuristischen Wert aufweisen würde als der bisher besuchte Zustand. Es gibt allerdings Suchszenarien bei denen die Heuristik nicht weiterhilft um einem Ziel näher zu kommen. Dies ist insbesondere dann der Fall, wenn sich der heuristische Wert von benachbarten Zuständen nicht ändert. Für die gierige Bestensuche würde das bedeuten, dass die Suche auf *Plateaus* und somit blind verläuft, weil sich dieser Suchalgorithmus ausschliesslich auf die Heuristik stützt. Algorithmen, die die Heuristik als Wegweiser verwenden, gehören zur Klasse der *heuristischen* Suchalgorithmen.

In dieser Arbeit geht es darum, in Fällen wie den Plateaus trotzdem eine Orientierung im Zustandsraum zu haben, indem Zustände neben der Heuristik einer weiteren Priorisierung unterliegen. Die hier vorgestellte Methode nutzt Abhängigkeiten zwischen Operatoren aus und erweitert die gierige Bestensuche. Wie stark Operatoren voneinander abhängen, betrachten wir anhand eines Abstandsmasses, welches vor der eigentlichen Suche berechnet wird. Die grundlegende Idee ist, Zustände zu bevorzugen, deren Operatoren im Vorfeld voneinander profitierten. Die Heuristik fungiert hierbei erst im Nachhinein als Tie-Breaker, sodass wir einem vielversprechenden Pfad zunächst folgen können, ohne dass uns die Heuristik an einer anderen, weniger vielversprechenden Stelle suchen lässt.

Die Ergebnisse zeigen, dass unser Ansatz in der reinen Suchzeit je nach Heuristik performanter sein kann, als wenn man sich ausschliesslich auf die Heuristik stützt. Bei sehr informationsreichen Heuristiken kann es jedoch passieren, dass die Suche durch unseren Ansatz eher gestört wird. Zudem werden viele Probleme nicht gelöst, weil die Berechnung der Abstände zu zeitaufwändig ist.

Inhaltsverzeichnis

1 Einleitung

In der klassischen Handlungsplanung geht es im Wesentlichen darum mit einem Agenten ein Planungsproblem zu lösen. Beginnend in einem initialen Zustand sucht sich der Agent einen Weg durch den vom Planungsproblem induzierten Zustandsraum zu einem Zielzustand, indem er sukzessiv Operatoren anwendet. Nach Anwendung eines Operators erfolgt der Wechsel von einem Zustand in den Nächsten. Die dabei entstehende Sequenz von Operatoren wird auch *Plan* genannt und stellt die Lösung eines Planungsproblems dar.
Eine bewährte Methode für die Durchsuchung eines Zustandsraumes ist die heuristische Suche, welche die *Heuristik* als zusätzliche, richtungsweisende Information verwendet. Sie schätzt für jeden Nachfolge-Zustand den Abstand zum Ziel und weist diesem Zustand einen numerischen Wert zu. Algorithmen, wie die gierige Bestensuche, verwenden für die Wahl des nächsten zu besuchenden Zustands ausschliesslich die Heuristik. Die Intuition ist während der Suche den Zustand zu besuchen, der den geringsten heuristischen Wert aufweist. Ein bekanntes Problem ist das Suchen auf Plateaus, was der blinden Suche gleicht. Dies tritt auf, wenn die heuristischen Werte der benachbarten Zustände identisch sind.
In dieser Arbeit versuchen wir dieses Problem zu dämpfen, indem wir neben der Heuristik ein weitere Priorisierung der Nachfolge-Zustände vornehmen. Dafür erweitern wir die gierige Bestensuche und betrachten die entstandenen Ergebnisse im Rahmen der klassischen Handlungsplanung. Die vorgestellte Herangehensweise stammt aus dem Bereich des Model Checkings und wurde erstmals von Kupferschmid und Wehrle eingeführt [1].
Unser Ansatz untersucht aufeinanderfolgende Operatoren auf gegenseitigen Nutzen. Wir führen ein Abstandsmass zwischen Operatoren ein, das zeigt, wie stark dieser gegenseitige Nutzen ist. Beim Durchlaufen eines Zustandsraumes werden dann jene Operator-Sequenzen bevorzugt ausgeführt, deren Operatoren einen endlichen Abstand zueinander haben, d.h. voneinander profitieren. Die Idee ist so möglichst lange einem Pfad zu folgen bis Teilziele erreicht sind. Erst dann sucht der Algorithmus an einer anderen, rein von der Heuristik bestimmten Stelle weiter. Bis dahin unterliegen Nachfolge-Zustände vorrangig einer von den Operatoren bestimmten Priorisierung.
Bevor der nächste Zustand besucht wird, betrachten wir, ob der Operator, der zum diesem Zustand führt einen Nutzen aus dem vorangegangenen Operator ziehen kann. Beispielsweise könnte der vorangegangene Operator durch seinen Effekt Variablen derartig setzen, dass

die Vorbedingung des aktuell untersuchten Operators dadurch erfüllt wäre. Wir sprechen hierbei von *Interferenz* zweier Operatoren. Das eben erwähnte Beispiel zeigt sogar direkte Interferenz. Eine weitere Möglichkeit wäre, wenn ein Operator indirekt, über einen oder mehrere andere Operatoren interferieren würde. Zum Beispiel gebe es drei Operatoren a, b und c, wobei a mit b und b wiederum mit c interferiert, allerdings keine direkte Interferenz zwischen a und c existiert. Dann stehen a und c zwar noch in einer Beziehung, aber über einen Umweg. Der Abstand zwischen a und c wäre grösser. Je grösser der Abstand zwischen zwei Operatoren, desto weniger hoch wird der Nachfolge-Zustand priorisiert. Ein Zustand wird *abgeschnitten*, d.h. nicht mehr betrachtet, sobald die beiden vorherigen Operatoren in keiner Weise interferieren. Sie haben dann einen unendlichen Abstand zueinander und arbeiten auf völlig disjunkten Teilproblemen. Die konkrete Umsetzung dieser Herangehensweise geschieht über das von Helmert entwickelte Fast-Downward Planungssystem [2]
Im vierten Kapital stellen wir Ergebnisse vor, die bei der Anwendung der kontext-basierten Methode auf STRIPS-Domänen entstanden. Diese verglichen wir mit den Ergebnissen der gierigen Bestensuche. Für den Vergleich benutzten wir drei Konfigurationen. Zunächst liessen wir blind suchen, anschließend mit zwei Heuristiken, die einen unterschiedlichen starken Informationsgehalt besitzen. Es hat sich gezeigt, dass die blinde gierige Bestensuche ungezielter sucht als unsere Methode, die einen Nutzen aus der Untersuchung der Operatoren-Verhältnisse zieht. Unter Verwendung der Heuristiken kann die Suche jedoch gezielter sein, wenn man ausschließlich der Heuristik folgt. Dies hängt vom Informationsgehalt der Heuristik ab. Bei der schwächeren Heuristik sahen wir bei unserer Methode eine Verbesserung der reinen Suchzeit (ohne Berechnung der Abstände), bei der Stärkeren war das Gegenteil der Fall. Die Berechnung der Abstände stellte sich als sehr zeitraubend heraus, sodass das rein Heuristik-basierte Suchverfahren deutlich mehr Probleme löste als unsere Methode. Dafür zeigte unser Verfahren bei der blinden Suche eine höhere Anzahl gefundener Lösungen.

2

Präliminarien

Dieses Kapitel schafft einen Hintergrund vor dem die eigentliche Idee dieser Arbeit steht. Zunächst folgt eine formale Definition einer SAS^+-Planungsaufgabe [3]. Im darauf folgenden Abschnitt wird ein Basisalgorithmus der gierigen Bestensuche vorgestellt, den wir im dritten Kapital erweitern.

2.1 Planungsaufgabe nach SAS^+

Planungsaufgaben charakterisieren sich dadurch, dass ein Agent, ausgehend von einem Startzustand nach einem Zielzustand sucht. Er wendet dabei laufend Operatoren an, mit denen jeweils ein Wechsel von einem Zustand in den Nächsten vollzogen wird. Die Abfolge von Operatoren, die vom Startzustand zu einem Zielzustand führen, stellen eine Lösung der Planungsaufgabe dar. Planungsaufgaben werden mit einem Planungsformalismus einheitlich repräsentiert und lassen sich somit von Planungssystemen unabhängig von ihren Domänen lösen. Wir verwenden den SAS^+-Planungsformalismus. Der Agent wendet während seiner Suche Operatoren an, mit denen er neue Zustände generiert und anschliessend besucht. Zustände bestehen nach SAS^+ aus einer Menge von Variablen, denen ein Wert aus einem endlichen Wertebereich zugewiesen wurde. Angewandte Operatoren haben den Effekt diese Variablen im Rahmen des Wertebereichs zu setzen. Somit unterscheiden sich Zustände durch eine individuelle Belegung der einzelnen Variablen. Das Ziel ist nun, ausgehend von einer Startbelegung eine passende Sequenz von Operatoren zu finden, die Variablen entsprechend definierter Zielbelegungen setzen. Jeder Zielzustand muss mindestens die Zielbelegungen erfüllen.
Formal werden SAS^+-Planungsaufgaben als 4-Tupel aufgefasst.

Definition 1 (Planungsaufgabe)**.** *Eine Planungsaufgabe ist ein 4-Tupel* $\Lambda = \langle \mathcal{V}, \mathcal{O}, s_0, G \rangle$*:*

- $\mathcal{V}$ *sei eine endliche Menge von Zustandsvariablen, die Werte aus einem endlichen Wertebereiche annehmen können. Zustände bestehen aus einer Menge von Variable/Wert-Paaren bzw. Belegungen. Dabei unterliegt jede Variable aus* $\mathcal{V}$ *einer Wertzuweisung.*
- $\mathcal{O}$ *ist eine endliche Menge von Operatoren. Jeder Operator* $o \in \mathcal{O}$ *besteht wiederum aus Vorbedingungen (pre(o)), Effekten (eff(o)) und Kosten (cost(o)), die beim Anwenden*

von o entstehen. pre(o) und eff(o) sind jeweils Mengen von belegten Zustandsvariablen aus $\mathcal{V}$. Hierbei gilt, dass in pre(o) und eff(o) nicht alle Zustandsvariablen vertreten sein müssen. Sei s der derzeitig besuchte Zustand und sei $o \in \mathcal{O}$. Der Operator o ist genau dann anwendbar, wenn $pre(o) \subseteq s$ gilt. D.h. die Belegung in s genügt den Vorbedingungen pre(o) und o kann angewendet werden. Sei weiter s' der neue Zustand, den man nach Anwendung von o erreicht. So gilt für s' neben der Belegung von s auch zusätzlich alle Effekte von o. Effekte haben eine Um- bzw. Neubelegung der Variablen zufolge.

- *s_0 ist der Anfangszustand bestehend aus einer vollständigen Belegung aller Variablen aus $\mathcal{V}$.*
- *G ist eine Menge von Belegungen mit Zustandsvariablen aus $\mathcal{V}$. G kann partiell belegt sein, d.h. dass nicht alle Zustandsvariablen in G vertreten sein müssen. G beschreibt die zu erreichenden Ziele. Jeder Zielzustand muss mindestens die Belegung aus G beinhalten.*

Die Lösung einer gegebenen Planungsaufgabe wird auch Plan genannt und als Sequenz von sukzessiv anwendbaren Operatoren definiert.

Definition 2 (Plan)**.** *Sei Λ ein gegebenes Planungsproblem und ist λ ein Plan bestehend aus einer Sequenz von Operatoren aus $\mathcal{O}$. Durch sukzessives Anwenden der Operatoren aus λ gelangt man vom initialen Zustand in einen Zielzustand. Sei n die Anzahl der dafür benötigten Operatoren.*

$$\lambda = \langle o_1, o_2, ..., o_n \rangle$$

Der Operator o_1 sei im Startzustand anwendbar und sei o_n jener Operator mit dem man schliesslich in einen Zielzustand gelangt. Die Plankosten χ sind die Aufsummierung aller Kosten der Operatoren in einem Plan.

$$\chi = \sum_{i=1}^{n} cost(o_i)$$

Ein Plan ist *optimal*, wenn es keinen kostengünstigeren Plan zu einem Zielzustand gibt. Ein Plan ist *suboptimal*, wenn dieser nicht der Kostengünstigste ist.

2.2 Gierige Bestensuche

Die *gierige Bestensuche* gehört zu der Klasse der heuristischen Suchalgorithmen. Mit ihr kann ein Zustandsraum nach einem Zielzustand durchsucht werden. Bei der Wahl des nächsten zu besuchenden Zustands bezieht man sich hierbei ausschliesslich auf den heuristischen Wert. Das bedeutet, das jener Zustand als nächstes besucht wird, der von allen Nachfolge-Zuständen den geringsten heuristischen Wert besitzt. Im Kontext der heuristischen Suchalgorithmen ist die Heuristik eine mathematische Funktion h, die jedem Zustand einen numerischen Wert zuweist. Sie schätzt die verbleibenden Pfadkosten zum Ziel. Bei einer zulässigen Heuristik gilt, dass der heuristische Wert eines Zustands nicht überschätzt werden, d.h. grösser als der wahre Abstand zum Ziel sein darf. Zusätzlich gelten dann folgende Eigenschaften:

- h ist zielerkennend. D.h. es gilt $h(g) = 0$, wobei g ein Zielzustand ist.
- h ist sicher. D.h., wenn $h(s) = \infty$ gilt, so muss für den wahren Abstand zum Ziel $h^*(s) = \infty$ gelten.

Der nachfolgende Algorithmus zeigt die gierige Bestensuche. Als Vorlage dienten Vorlesungsfolien aus der Veranstaltung "Grundlagen der Künstlichen Intelligenz" von Prof. Helmert [4].

Algorithmus 1 Gierige Bestensuche

```
1:  openlist := new priority queue
2:  openlist.initialize()
3:  closedlist := { }
4:  while openlist not empty do
5:      node = openlist.getMinimum()
6:      if closedlist.contains(node.state) = none then
7:          closedlist.add(node.state)
8:          if node.state is a goalstate then
9:              return getPath(node)
10:         for each successor s' of node.state do
11:             if h(s') < ∞ then
12:                 openlist.add(createNode(s'))
13: return unsolveable
```

Die Grundidee ist solange neue Zustände zu besuchen bis einer ein Zielzustand ist oder es keine Zustände mehr gibt, die besucht werden könnten. Gilt Letzteres so existiert vom ausgehenden Anfangszustand keine Lösung. Welche Zustände besucht werden entscheidet dabei die Heuristik. Üblicherweise werden alle möglichen Kandidaten (Nachfolge-Zustände) in einer gemeinsamen Datenstruktur als *Suchknoten* gespeichert - die sogenannte Openlist (Zeile 1). Sie wird typischerweise als Min-Heap implementiert, sodass man ihr den Zustand mit kleinstem heuristischen Wert in konstanter Zeit entnehmen kann. Die Nachfolge-Zustände repräsentieren wir als Suchknoten. Suchknoten haben den Vorteil, das man nach Erreichen des Ziels auf einfache Weise den Lösungspfad extrahieren kann (Zeile 9). Sie enthalten neben dem Zustand und Operator, der zu diesem Zustand führte auch den vorherigen Suchknoten. So können wir sukzessiv alle besuchten Suchknoten durchgehen, die angewandten Operatoren betrachten und daraus den *Plan* erstellen. Bevor die Suche startet wird die Openlist mit dem Wurzelknoten, welcher den Anfangszustand beinhaltet, initialisiert (Zeile 2).
Um Duplikate zu eliminieren, bedienen wir uns einer weiteren Datenstruktur, die bereits besuchte Zustände speichert. Sie trägt den Namen *closedlist* (Zeile 3) und kann als Mengenstruktur von Zuständen aufgefasst werden. Zu Beginn einer jeden Iteration entnehmen wir der Openlist einen Suchknoten. Die Methode *getMinimum()* (Zeile 5) entnimmt jenen Suchknoten, dessen Zustand den kleinsten heuristischen Wert aufweist. Der Suchknoten wird verworfen, falls dessen Zustand bereits besucht wurde. Andernfalls folgt nun die Überprüfung, ob es sich um einen Zielzustand handelt (Zeile 8). Ist das ebenso nicht der Fall werden die Nachfolger erzeugt und ausgewertet. Zustände, dessen Heuristik unendlich betragen (Zeile 11) werden *abgeschnitten*, indem sie nicht in die Openlist eingefügt werden und demnach in der nächsten Iteration auch nicht entnommen werden können. Alle anderen Zustände werden als Suchknoten in die Openlist hinzugefügt.
Die gierige Bestensuche kann suboptimale Pläne liefern, weil bei der Wahl der Nachfolge-Zustände nicht die Operator-Kosten berücksichtigt werden. Somit kann der Plan beliebig teuer sein. In der Arbeit akzeptieren wir auch suboptimale Pläne.

Im dritten Kapitel wird dieser Algorithmus wiederverwendet und um eine mächtigere Openlist erweitert. Die hier vorgestellte Openlist zeigt Schwächen bei Zuständen, deren $h(s)$ gleich ist zu anderen Zuständen. Unsere Methode bietet eine weitere Unterscheidung dieser Zustände.

3

Der Kontext-basierte Suchalgorithmus

In diesem Kapitel wird unser kontext-basierte Suchalgorithmus vorgestellt. Die grundlegende Idee kann mithilfe des Logistik-Problems illustriert werden. Hierbei geht es um Transportfahrzeuge, deren Ziel es ist Pakete von einer Startposition zu einer Zielposition zu transportieren. Seien die Fahrzeuge in drei Städten A, B und C mit je vier Positionen verteilt. In jeder Stadt definieren wir für jeweils ein Fahrzeug die Startposition und die Zielposition. In Abbildung 3.1 wurde die Startposition hell und die Zielposition umkringelt markiert. Das Gesamtziel des Logistik-Problems ist, dass jedes dieser Fahrzeuge sein Paket über Zwischenstationen in die Zielposition befördert. So müsste beispielsweise das erste Fahrzeug in Stadt A ausgehend von Position 1 über Position 2 und 3 nach Position 4 fahren um sein Ziel zu erreichen. Das Fahrzeug hätte theoretisch die Möglichkeit zurückzufahren, auch wenn das Paket noch nicht zugestellt wurde. Dies gilt äquivalent auch für die Fahrzeuge in Stadt B und C.

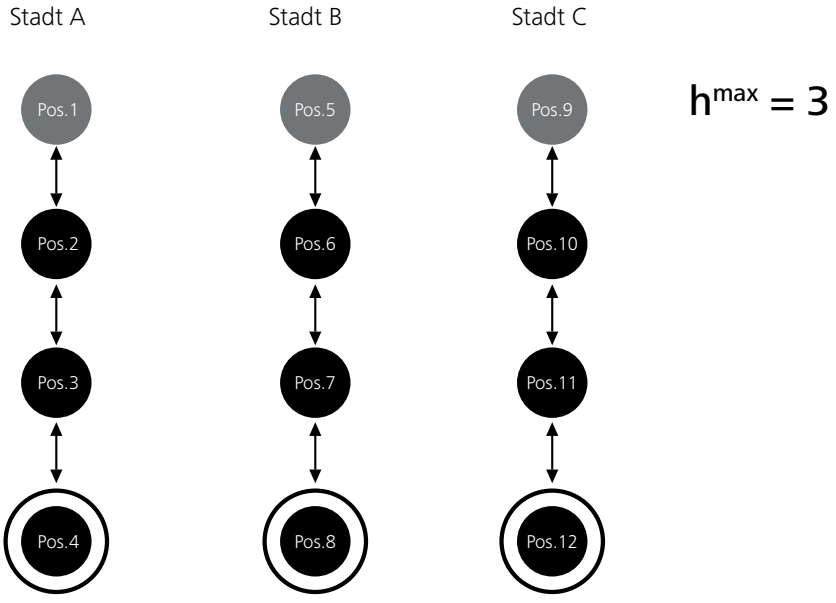

Abbildung 3.1: Logistik-Problem mit anfänglich drei Transportfahrzeuge

Die angewandte Heuristik sei h^{max}[5] und jeder Operator hätte Kosten von eins. Das Anwenden eines Operator entspricht der Fahrt von einem Fahrzeug von einer Position in eine Andere. h^{max} verwendet als Schätzung für die restliche Pfadlänge die maximalen Pfadkosten in einem relaxierten Planungsgraphen. Da die einzelnen Teilprobleme unabhängig voneinander sind, ergibt h^{max} für den Startzustand eine Schätzung von drei. Selbst wenn eines der Fahrzeuge in Richtung seines Ziels fährt, bleibt h^{max} gleich (siehe Abbildung 3.2). Dies liegt daran, dass im relaxierten Planungsgraphen die Pfadkosten für die anderen beiden Teilprobleme (Stadt B und C) weiterhin drei betragen. Erst wenn alle drei Fahrzeuge die erste Zwischenstation erreichen, zeigt sich eine Änderung der Heuristik. In diesem Beispiel ist der Informationsgehalt der Heuristik eher gering. Bei Verwendung der gierigen Bestensuche mit der h^{max}-Heuristik treten aufgrund des gleichbleibenden, heuristischen Wertes der Nachfolge-Zustände Plateaus auf. Dies könnte zur Folge haben, dass Zustände exploriert bzw. generiert werden, die nicht nötig wären. Zum Beispiel könnte das Fahrzeug in Stadt B in Position 6 und anschliessend das Fahrzeug in Stadt A wieder zurück in Ausgangsposition fahren. Dies wäre ein neuer, aber für die Lösung des Problems unwichtiger Zustand. h^{max} wäre über den gesamten Vorgang drei, weil das Fahrzeug in Stadt C in der Ausgangsposition verweilt.

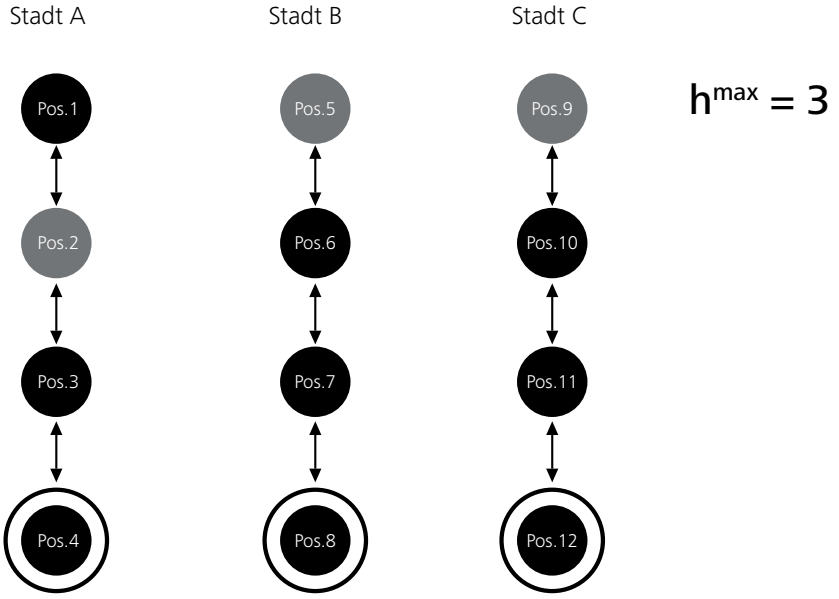

Abbildung 3.2: Logistik-Problem. Erreichen der ersten Zwischenstation zeigt gleichbleibende Heuristik (hmax)

Unser Ansatz erkennt die Beziehung zwischen den Operatoren entlang eines Pfades und wendet diese Operatoren vorzugsweise an. Angenommen wir befinden uns im Anfangszustand wie Abbildung 3.1 zeigt. Von hier ausgehend kann ein beliebiger Operator angwandt werden. Beispielsweise wenden wir den Operator an, der das Fahrzeug in Stadt A von Position 1 nach Position 2 bringt, wie Abbildung 3.2 zeigt. Der eben angwandte Operator setzt eine Zustandsvariable derartig, dass die Vorbedinung des Operators erfüllt ist, der das Fahrzeug weiter nach Position 3 befördern kann. Für die anderen Operatoren, die anwendbar wären (für Fahrzeuge in Stadt B und C), gilt dieser Zusammenhang nicht. Somit wenden wir sukzessive die Operatoren entlang eines Pfades an, bis ein Teilziel erreicht wurde bzw. bis das

Fahrzeug in Stadt A sein Ziel erreicht hat. Anschließend legt man den Fokus der Suche auf die anderen beiden Teilprobleme.

3.1 Interferenz zweier Operatoren

In diesem Abschnitt wird die Idee der Interferenz zwischen zwei Operatoren formalisiert.

Definition 3 (Interferenz)**.** *Sei Λ eine Planungsaufgabe und seien o und o' jeweils Operatoren aus $\mathcal{O}$. Wir sagen, dass o und o' miteinander interferieren, wenn einer der folgenden Fälle zutrifft:*

- *In pre(o) kommt mindestens eine Variable aus $\mathcal{V}$ vor, die in eff(o') gesetzt wird.*
- *In eff(o) wird mindestens eine Variable aus $\mathcal{V}$ gesetzt, die in pre(o') vorkommt.*
- *In eff(o) wird mindestens eine Variable aus $\mathcal{V}$ gesetzt, die auch in eff(o') gesetzt wird.*

Diese Interferenz-Regeln wurden erstmals von Kupferschmid und Wehrle aufgestellt [1] und im Model Checking eingesetzt. Sie werden in dieser Arbeit nun im Rahmen der klassischen Handlungsplanung erneut aufgegriffen. Die Intuition ist, dass Operatoren, die mit gemeinsamen Zustandsvariablen zu tun haben, voneinander profitieren und es sinnvoll sein kann diese Operatoren nacheinander anzuwenden, anstatt an einer anderen Front zu suchen.

3.2 Interferenz-Distanz

Die angestrebte Priorisierung von Nachfolge-Zuständen, hängt von den zuvor angewandten Operatoren ab. Um die Priorisierung zuverfeinern, führen wir in diesem Abschnitt ein Abstandsmass zwischen Operatoren ein.

Definition 4 (Interferenz-Distanz)**.** *Sei $n(o, o')$ eine Funktion, die für zwei gegebene Operatoren o und o' einen numerischen Wert d zurückgibt ($d \in \mathbb{N}$). Den Wert d nennen wir die Interferenz-Distanz zwischen zwei Operatoren o und o' aus $\mathcal{O}$. Wir treffen folgende Unterscheidungen:*

- *$d = 0$, wenn $o = o'$ gilt.*
- *$d =$ minimale Anzahl der paarweisen Interferenzen gdw. $o \neq o'$ und es eine Kette von paarweisen Interferenzen zwischen Operatoren aus $\mathcal{O}$ gibt, sodass o der Anfang und o' das Ende dieser Kette ist. Es könnte verschieden lange Ketten von paarweisen Interferenzen zwischen o und o' geben. Wir fordern die kleinstmögliche Anzahl von paarweisen Interferenzen zwischen o und o'.*
- *$d = \infty$, wenn o und o' nicht miteinander interferieren, auch nicht über andere Operatoren.*

Die Idee ist, je nach Grösse einer Kette von paarweisen Interferenzen zwischen zwei Operatoren, die Priorisierung eines Nachfolge-Zustands entsprechend anzupassen. Im nächsten Abschnitt führen wir für jede auftretende Interferenz-Distanz eine eigene Priorisierungs-Stufe ein. Um für jeden Operator die Interferenz-Distanz zu allen anderen Operatoren aus $\mathcal{O}$ zu speichern, verwenden wir als Datenstruktur eine Distanzmatrix, die wie folgt aufgebaut ist.

Definition 5 (Distanzmatrix)**.** *Sei k die Anzahl aller Operatoren aus $\mathcal{O}$, dann ist die Distanzmatrix D eine $k \times k$-Matrix, wobei jeder Eintrag aus der Interferenz-Distanz zweier Operatoren besteht.*

$$D = \begin{array}{c} \\ o_1 \\ o_2 \\ \vdots \\ o_k \end{array} \begin{array}{c} \begin{array}{cccc} o_1 & o_2 & \dots & o_k \end{array} \\ \begin{pmatrix} 0 & n(o_1,o_2) & \dots & n(o_1,o_k) \\ n(o_2,o_1) & 0 & \dots & n(o_2,o_k) \\ \vdots & \vdots & \ddots & \vdots \\ n(o_k,o_1) & n(o_k,o_2) & \dots & 0 \end{pmatrix} \end{array}$$

3.3 Algorithmus und Implementierung

In diesem Abschnitt stellen wir unseren kontext-basierten Suchalgorithmus vor und gehen auf dessen Implementierung ein. Im Wesentlichen besteht er aus zwei Teilen - die anfängliche Berechnung der Distanzmatrix und der eigentlichen Suche, wofür wir die gierige Bestensuche aus Abschnitt 2.2 derartig erweitern, dass die Interferenz-Distanzen in die Priorisierung der Nachfolge-Zustände vorrangig einbezogen werden.

3.3.1 Berechnung der Distanzmatrix

Für die Berechnung der Distanzmatrix verwenden wir den Floyd-Algorithmus [6], der die kürzesten Pfade zwischen zwei Operatoren berechnet. Dafür wird die Distanzmatrix initialisiert. Im Prinzip detektieren wir alle direkten Interferenzen zwischen zwei Operatoren und notieren uns an entsprechender Stelle eine Interferenz-Distanz von eins. Operatoren-Paare die nicht direkt interferieren erhalten zunächst einen unendlichen Wert.

Algorithmus 2 Floyd-Algorithmus: Initialisierung der Distanzmatrix

```
1: D := new k × k Matrix with ∞ in all entries        ▷ k sei die Anzahl aller Operatoren
2: for each operator i ∈ O do
3:     for each operator j ∈ O do
4:         if iisequaltoj then
5:             D(i, j) = 0                              ▷ 0 auf der Diagonale
6:         else if i interferes with j then
7:             D(i, j) = 1
```

Der Floyd-Algorithmus berechnet nun die kürzesten Pfade der Operatoren-Paare. Somit beinhaltet die Distanzmatrix nur minimale Abstände zwischen Operatoren, wie Definition 4 fordert.

Algorithmus 3 Floyd-Algorithmus: Berechnung der kürzesten Pfade

```
1: for each operator k ∈ O do
2:     for each pair of operators i, j ∈ O do
3:         D(i, j) = min(D(i, j), D(i, k) + D(k, j))
```

Die Komplexität des Floyd-Algorithmus' ist in $O(n^3)$, wobei n die Anzahl der Operatoren ist.

3.3.2 Suchalgorithmus

In diesem Abschnitt erweitern wir den Suchalgorithmus der gierigen Bestensuche mit unserer kontext-basierten Suche, bekannt aus dem Model Checking [1]. Die Idee ist einen Nachfolge-Zustand anhand der Interferenz-Distanz der beiden vorherigen Operatoren zu priorisieren. Um die Interferenz-Distanzen bei der Wahl des nächsten zu besuchenden Zustands einfliessen zu lassen, verwenden wir eine erweiterte Openlist und überschreiben ihre Funktionen zum Entnehmen und Einfügen von Suchknoten. Ansonsten werden am Algorithmus 1, den wir im Abschnitt 2.2 vorstellen, keine Änderungen vorgenommen. Unsere neue Openlist besteht aus $N + 1$ Prioritätsschlangen, wobei N die maximal berechnete Interferenz-Distanz ist. Die Idee ist für jede, in der Distanzmatrix vorkommende Interferenz-Distanz eine eigene Prioritätsschlange zu benutzen. Ausgenommen davon ist die unendliche Interferenz-Distanz. So unterliegt jeder neu eingefügte Suchknoten je nach Interferenz-Distanz einer Priorisierung. Beispielsweise sind Suchknoten aus der Prioritätsschlange, die für eine Interferenz-Distanz von null steht, höher priorisiert als Suchknoten aus einer Prioritätsschlange, stehend für eine Interferenz-Distanz von zwei. Ob und in welche Prioritätsschlange ein Suchknoten eingefügt wird, hängt von den beiden vorangegangen Operatoren ab, die zum Zustand des Suchknotens führen. Im Folgenden wird die Funktion zum Hinzufügen von einem Suchknoten n in die Openlist beschrieben.

Algorithmus 4 Funktion: add(Node n)

```
1  priority-queues := given N + 1 priority queues ▷ N sei die maximale Interferenz-Distanz
2  o := operator to current state s                ▷ s sei der aktuelle Zustand
3  o' := applicable operator to switch state from s to n.state
4  if o not writes variable corresponding to G then   ▷ G sei die Zielbelegung
5      if D(o, o') ≠ ∞ then                ▷ D sei die zuvor berechnete Distanzmatrix
6          priority-queues[D(o, o')].add(n)
7      else
8          prune n.state
9  else
10     priority-queues[0].add(n)
```

Sei s der aktuell besuchte Zustand. Die *add(Node n)*-Funktion wird aufgerufen, um den Nachfolge-Zustand $n.state$ in Form des Suchknotens n der Openlist hinzuzufügen. Dabei sei o der Operator, der zu s führte und sei o' der, in s anwendbarer Operator, der zu $n.state$ führt. In *add(Node n)* betrachten wir das Verhältnis der beiden vorherigen Operatoren o und o'. Zunächst wird überprüft, ob o Variablen entsprechend der Zielbelegung gesetzt hat (Zeile 4). Das würde bedeuten, dass wir ein Teilziel über o erreicht haben und den Pfad, der zu diesem Teilziel führte, beendet ist. Stattdessen priorisieren wir den Nachfolge-Zustand $n.state$ hoch, indem n in die höchst priorisierte Schlange eingefügt wird, unabhängig von der Interferenz-Distanz zwischen o und o'. Die Idee ist einen komplett neuen Pfad zu beginnen. Im einleitenden Beispiel von diesem Kapitel, würde dies bedeuten, dass eines der Fahrzeuge sein Ziel erreicht und nun das nächste Fahrzeug beginnt in Richtung des Ziels zu fahren. Falls wir uns noch innerhalb eines Teilpfades aufhalten und o keine Zielvariablen setzt, betrachten wir die Interferenz-Distanz zwischen o und o', welche in der zuvor berechneten Distanzmatrix abgelegt ist (Zeile 5). Falls die Interferenz-Distanz unendlich beträgt, arbeiten die beiden Operatoren o und o' auf disjunkten Teilproblemen. D.h., dass die Variablen in pre(o') und eff(o') im gesamten Teilproblem nicht benötigt werden. In Bezug auf das

Logistik-Problem gibt es drei Teilprobleme und wir folgen beispielsweise den Pfad, der das Fahrzeug in Stadt A ins Ziel befördet. Der Operator o' arbeitet bei unendlicher Interferenz-Distanz zu o auf Variablen, die für die Fahrzeuge in Stadt B oder C gedacht sind. Somit können wir den Zustand $n.state$, den wir durch o' erreicht hätten, abschneiden bzw. nicht in die Openlist einfügen. Andernfalls könnte der Suchalgorithmus an einer anderen Stelle weitersuchen und den Fokus auf den aktuell untersuchten Pfad verlieren, wenn unendliche Interferenz-Distanzen nicht berücksichtigt werden. Falls die Interferenz-Distanz zwischen o und o' endlich ist, fügen wir die $n.state$ in die entsprechende Prioritätsschlange ein.
Das Entnehmen eines Suchknotens geschieht in zwei Schritten. Ausgehend von der höchst priorisierten Schlange suchen wir sukzessive alle Prioritätsschlangen nach vorhandenen Suchknoten ab, bis eine nicht-leere Prioritätsschlange gefunden wurde. Wenn beispielsweise die nullte Prioritätsschlange keine Suchknoten enthält, untersuchen wir die Erste nach Suchknoten usw. . Nach erfolgreicher Bestimmung der Prioritätsschlange, entnehmen wir ihr einen Suchknoten, dessen Zustand, von allen in der Schlange befindlichen Zustände, den geringsten heuristischen Wert aufweist. Somit unterscheiden wir Nachfolge-Zustände zunächst nach der Interferenz-Distanz der beiden vorherigen Operatoren und erst im zweiten Schritt nach dem heuristischen Wert. Die Heuristik fungiert demnach als Tie-Breaker. Tie-Breaker kommen dann zum Einsatz, wenn die erste Priorisierungsmethode nicht ausreicht, um den nächstbesten zu besuchenden Zustand eindeutig zu bestimmen. Dies wäre bei uns der Fall, wenn sich in einer Prioritätsschlange mindestens zwei Suchknoten aufhalten würden. Die beiden vorherigen Operatoren dieser Zustände haben somit die gleiche Interferenz-Distanz zueinander. Die Heuristik macht anschließend eine eindeutige Wahl des Nachfolgers eher möglich. Die konkrete Implementierung erfolgte über das Fast-Downward Planungssystem [2].

4

Experimentelle Analyse

In diesem Kapitel stellen wir Ergebnisse vor, die beim Anwenden der kontext-basisierten Suche (KBS) auf STRIPS-Domänen entstanden und interpretieren diese. Zudem vergleichen wir unsere Herangehensweise mit der reinen gierigen Bestensuche (GBS) unter Verwendung verschiedener Heuristiken. Im ersten Abschnitt setzen wir Rahmenbedingungen für die Experimente. Der letzte Abschnitt zeigt die Ergebnisse aus mehreren Perspektiven.

4.1 Rahmenbedingungen

Die Experimente beruhen auf STRIPS-Domänen, die im Fast-Downward Planungssystem enthalten sind. Bevor die eigentliche Suche startet, werden die in STRIPS formulierten Planungsprobleme in SAS^+-Probleme konvertiert. STRIPS-Planungsprobleme sind ähnlich aufgebaut, wie die in SAS^+, mit dem Unterschied, dass Variablen in $\mathcal{V}$ nur boolesche Werte annehmen können und Zustände als eine Menge von, auf *wahr* gesetzten Variablen betrachtet werden. Wir beschränken uns auf STRIPS-Probleme, weil unsere Methode *bedingte Effekte* in Operatoren nicht unterstützt.
Jede Probleminstanz einer Domäne erhielt eine maximale Zeit von 30 Minuten und 2 GB Speicher, um einen Lösungspfad zu finden. Die Suche wurde von auf Intel Xeon E5-2660 (2.2 GHz) Prozessoren ausgeführt. Probleme die den Zeit- oder Speicherrahmen überschritten, gelten als *ungelöst*. Wir untersuchten die KBS und verglichen sie mit der GBS aus Abschnitt 2.2 unter drei Konfigurationen. Zunächst liessen wir beide Algorithmen uninformativ, d.h. ohne Heuristik suchen. Die beiden anderen Konfiguration benutzten zum einen die aus Kapitel 3 bekannte h^{max}- und zum anderen die bewährte h^{FF} [7] Heuristik, die erstmals von Hoffmann und Nebel eingeführt wurde.

4.2 Ergebnisse

In diesem Abschnitt stellen wir Ergebnisse vor, die beim Vergleich zwischen der KBS und GBS entstanden. Für beide Suchalgorithmen suchten wir blind, später dann informiert mit der h^{max}- und h^{FF} Heuristik.

4.2.1 Ergebnisse mit blinder Suche

Die folgende Tabelle liefert einen Überblick einiger wichtiger Attribute im uninformierten Fall über alle Domänen. Die verwendeten Domänen werden erstmals in Tabelle 4.2 aufgelistet.

Überblick	**GBS**	**KBS**
Gelöste Probleme - Summe	260	**288**
Evaluationen - im geom. Mittel	78408.44	**36075.39**
Expansionen - im geom. Mittel	61134.44	**15744.18**
Generierte Knoten - im geom. Mittel	379596.23	**99261.75**
Vorberechnungszeit - im geom. Mittel	**0.10**	1.02
Reine Suchzeit - im geom. Mittel	1.04	**0.36**
Gesamte Suchzeit - im geom. Mittel	1.04	**0.87**

Tabelle 4.1: Überblick: blinde Suche, Vergleich GBS und KBS

Hierbei ist deutlich zu sehen, dass die Berücksichtigung der Interferenz zwischen Operatoren in der KBS einen gewissen Informationsgehalt bietet um ein Planungsproblem gezielter zu lösen. Unser Suchalgorithmus braucht anfänglich zwar eine hohe Vorberechnungszeit für die Berechnung der Distanzmatrix und verbraucht somit effektive Suchzeit, kann jedoch die verbleibende Suchzeit besser nutzen als die blinde GBS. Das spiegelt sich auch deutlich in der Anzahl der Expansionen, Evaluationen und Generierungen von Suchknoten wider. KBS sucht gezielter nach einer Lösung. Die reine Suchzeit ist im geom. Mittel bei der KBS um das 2.88-fache geringer als bei der GBS. Abbildung 4.1 zeigt die benötigte *reine* Suchzeit für die jeweiligen Probleminstanzen, wobei jeder Punkt eine Probleminstanz repräsentiert. Unter reiner Suchzeit verstehen wir die Suchzeit ohne die anfängliche Vorberechnung. Die Punkte unterhalb der Geraden zeigen, dass KBS in den meisten Fällen gezielter sucht als GBS. In der uninformierten Suche scheint es demnach durchaus sinnvoll zu sein die Verhältnisse der Operatoren im Vorfeld zu berechnen, bevor die eigentliche Suche startet. Allerdings sehen wir Optimierungsbedarf bei der Berechnung der Distanzmatrix.

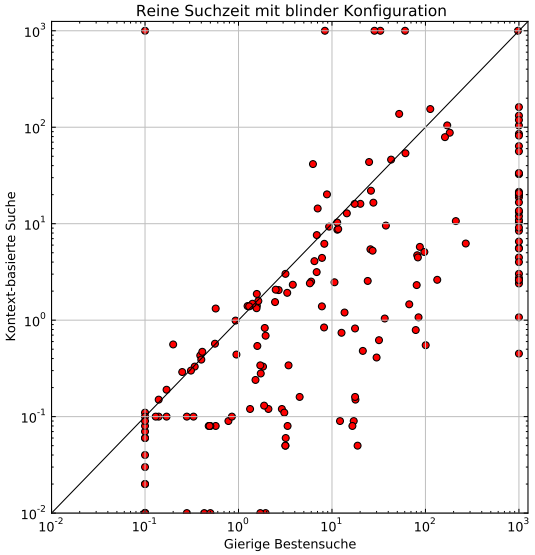

Abbildung 4.1: Streuungsdiagramm: Vergleich der reinen Suchzeiten zwischen GBS und KBS

Kosten	**GBS**	**KBS**
blocks (19)	**304**	306
driverlog (7)	**96**	212
elevators-sat08-strips (1)	**71**	210
freecell (14)	**172**	263
grid (1)	**14**	**14**
logistics00 (10)	**194**	322
logistics98 (2)	**33**	92
miconic (50)	**926**	1151
nomystery-sat11-strips (3)	**57**	58
openstacks-sat08-strips (6)	**15**	39
openstacks-strips (7)	**206**	**206**
pegsol-sat11-strips (17)	**143**	148
pipesworld-notankage (13)	**171**	304
pipesworld-tankage (8)	**101**	113
psr-small (48)	**838**	848
satellite (5)	**70**	78
scanalyzer-sat11-strips (1)	**36**	38
sokoban-sat08-strips (11)	**301**	**301**
sokoban-sat11-strips (3)	**83**	**83**
tpp (6)	**82**	98
transport-sat08-strips (6)	**2723**	3115
trucks-strips (6)	**121**	140
woodworking-sat08-strips (4)	870	940
Summe (248)	**7627**	9079

Tabelle 4.2: Pfadkosten der einzelnen Domänen mit blinder Konfiguration

In Bezug auf Plankosten schneidet unsere Methode jedoch sehr viel schlechter ab, wie Tabelle 4.2 zeigt. Dies liegt wahrscheinlich daran, dass jene Nachfolge-Zustände bevorzugt besucht werden, deren Operatoren höhere Kosten besitzen. Die Kosten eines Operators werden bei beiden Methoden nicht berücksichtigt.

4.2.2 Ergebnisse mit informierter Suche

Auch in diesem Abschnitt folgt zunächst eine holistische Auswertung einiger Attribute, die während der Suche über alle Domänen entstanden. Hierbei betrachten wir speziell Ergebnisse, die bei der heuristischen Suche auftraten. Deutlich zu sehen sind jeweils die Unterschiede bei der Anzahl der gelösten Planungsinstanzen. Die geringe Anzahl auf Seiten der KBS liegt

Überblick	h^{FF}**-GBS**	h^{FF}**-KBS**	h^{max}**-GBS**	h^{max}**-KBS**
Gelöste Probleme - Summe	**764**	492	**458**	416
Evaluationen - im geom. Mittel	**377.54**	1783.43	**4208.74**	4442.11
Expansionen - im geom. Mittel	**90.98**	535.90	1388.40	**1331.32**
Generierte Knoten - im geom. Mittel	**591.55**	3613.31	10007.14	**9354.75**
Vorberechnungszeit - im geom. Mittel	**0.10**	1.02	**0.10**	1.02
Reine Suchzeit - im geom. Mittel	**0.13**	0.17	0.33	**0.27**
Gesamte Suchzeit - im geom. Mittel	**0.13**	0.60	**0.33**	0.67

Tabelle 4.3: Überblick: heuristische Suche mit h^{FF} und h^{max} für jeweils GBS und KBS

vermutlich vorwiegend daran, dass die Vorberechnungszeit der Distanzmatrix so hoch ist, dass in den 30 Minuten pro Probleminstanz keine Zeit mehr für die reine Suche übrig bleibt. Die Distanzmatrix berechnen wir, wie Abschnitt 3.3.1 zeigt mit Floyd's Algorithmus, welcher in $O(n^3)$ liegt, wobei n die Anzahl der Operatoren ist. Bei komplexeren Problemen mit einer grossen Anzahl von Operatoren, verbraucht die Berechnung erhebliche Zeit, wie die Attribute „Gelöste Probleme“ und „Vorberechnungszeit“ in Tabelle 4.3 zeigen. Zudem zeigt die reine Suchzeit, dass je informierter eine Heuristik ist, desto geringer ist der Nutzen, den man durch das Betrachten der Interferenz-Distanzen erhält. Im Fall der h^{max} Heuristik ist die reine Suchzeit der KBS nur geringfügig besser. Bei der gehaltvolleren h^{FF} Heuristik scheint die Verwendung unserer Priorisierungsmethode sogar eher kontraproduktiv zu sein. Die KBS führt bei informationsreicher Heuristik (wie FF) dazu, dass wir an Stellen suchen, die weniger zielführend sind, als wenn wir uns nur auf die Heuristik verlassen würden. In Abschnitt 5.1 stellen wir in wenigen Worten vor, wie dieses Problem evtl. zu bändigen ist.
Die Berechnung der Interferenz-Distanzen könnte auf verschiedene Arten optimiert werden. Einerseits können wir die maximale Interferenz-Distanz beschränken. Die Tabelle 4.4 zeigt die maximale Interferenz-Distanz je Domäne. Die Idee ist eine Grenze für die maximale Interferenz-Distanz vor Berechnung der Distanzmatrix festzulegen und Berechnungen abzubrechen, sobald dieser Wert überschritten wird. Somit würden grössere Interferenz-Distanz grob approximiert werden. Betroffene Nachfolge-Zustände fügen wir in solchen Fällen in die am wenigsten priorisierte Schlange ein. Eine weitere Optimierung ist das Berechnen von Interferenz-Distanzen zwischen Operatoren, die während der Suche auch wirklich betrachtet werden. Die Berechnung der Distanzmatrix ist demnach partiell und wird auf in die eigentliche Suche verlagert und nicht mehr vor einer Suche durchgeführt.

Max. Interferenz-Instanz	**KBS**
blocks (36)	36
driverlog (20)	54
elevators-sat08-strips (30)	72
freecell (80)	70
grid (5)	6
logistics00 (28)	84
logistics98 (35)	63
miconic (150)	295
nomystery-sat11-strips (20)	32
openstacks-sat08-strips (30)	72
openstacks-strips (30)	55
pegsol-sat11-strips (20)	20
pipesworld-notankage (50)	116
pipesworld-tankage (50)	53
psr-small (50)	98
satellite (36)	97
scanalyzer-sat11-strips (20)	10
sokoban-sat08-strips (30)	30
sokoban-sat11-strips (20)	20
tpp (30)	56
transport-sat08-strips (30)	45
trucks-strips (30)	32
woodworking-sat08-strips (30)	87

Tabelle 4.4: Maximal errechnete Interferenz-Distanz pro Domäne

5 Schlussfolgerung

In dieser Arbeit wurde die kontext-basierte Suche beruhend auf der gierigen Bestensuche vorgestellt. Im Rahmen der klassischen Handlungsplanung zeigte sich, dass die Berücksichtigung der Interferenz zwischen Operatoren als zusätzliche Wegweisung in einem Zustandsraum dienen kann. Jedoch ist die anfängliche Berechnung der Distanzmatrix in einigen Fällen zu zeitintensiv, sodass effektive Suchzeit geraubt wird und manche Probleme dadurch nicht mehr gelöst werden können. Die Vermutung ist, dass einige, bereits erwähnte Optimierungsvorschläge bezüglich der Berechnung der Distanzmatrix, deutlich bessere Ergebnisse hervorbringen könnten.

5.1 Ausblick

In Zukunft wäre es sicherlich interessant zu testen, inwiefern eine Verbesserung der Berechnung von Interferenz-Distanzen Auswirkungen auf die Suche hätte. Evtl. ist ein Verzicht auf die anfängliche Berechnung der *kompletten* Distanzmatrix ratsam, denn momentan berechnen wir auch Interferenz-Distanzen zwischen Operatoren, die während der Suche nie betrachtet werden. Für zukünftige Arbeiten könnte man die Berechnung in die eigentliche Suche verlagern, sodass die Interferenz-Distanz nur für wirklich betrachtete Operatoren berechnet wird. Zudem könnte man die Interferenz-Distanzen auf einen bestimmten Wert begrenzen, sodass die Berechnungen früher beendet sind. Nachfolge-Zustände, deren Operatoren einen größeren Abstand zueinander haben als die festgelegte Grenze, könnten in die am wenigsten priorisierte Schlange eingefügt werden.
Eine weitere Idee, die für interessante Ergebnisse sorgen könnte, wäre die Umkehrung der Priorisierungsmethoden. Wir verwendeten in dieser Arbeit die Interferenz-Distanz als erste Priorisierung von Nachfolge-Zuständen und erst anschließend die Heuristik als Tie-Breaker bzw. als zusätzliche Unterscheidung. Eine andere Möglichkeit wäre Nachfolge-Zustände zunächst nach dem heuristischen Wert und dann nach der Interferenz-Distanz der beiden vorangegangen Operatoren zu unterschieden. So könnten Suchalgorithmen eher von informationsreichen Heuristiken profitieren.

Literaturverzeichnis

[1] Wehrle, M. and Kupferschmid, S. Context-Enhanced Directed Model Checking. In van de Pol, J. and Weber, M., editors, *SPIN*, volume 6349 of *Proceedings of the 17th International SPIN Workshop*, pages 88–105. Springer (2010).

[2] Helmert, M. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246 (2006).

[3] Bäckström, C. and Nebel, B. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11:625–655 (1995).

[4] Helmert, M. 12. Klassische Suche: Bestensuche als Graphensuche. *Foliensätze der Veranstaltung "Grundlagen der Künstlichen Intelligenz"*, page 14 (2014). URL `http://ai.cs.unibas.ch/_files/teaching/fs14/ki/slides/ki12-handout4.pdf`.

[5] Bonet, B. and Geffner, H. Planning as Heuristic Search. *Artificial Intelligence*, 129(1):5–33 (2001).

[6] Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345 (1962).

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