Fachhochschule Wedel

Seminar

in der Fachrichtung

Technische Informatik

Themenbereich: Künstliche Intelligenz

Constraintsysteme

Vortrag Nr.: 6

Eingereicht von:	Stefan Schmidt
Erarbeitet im:	6. Semester
Abgegeben am:	01.06.2005
Referent:	Prof. Dr. Sebastian Iwanowski

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Überblick

Diese Arbeit befasst sich mit dem Thema Constraints bzw. Constraintsysteme. Generell handelt es sich dabei um ein System von Neben- und Randbedingungen, wobei es fast immer um die Frage geht, ob eine Konstellation gefunden werden kann, unter der alle Bedingungen erfüllt sind. In der Praxis treten solche Problemstellungen häufig auf, so dass viele Anstrengungen unternommen wurden, diesen Bereich von Problemstellungen theoretisch zu erfassen und effiziente Algorithmen zu entwickeln. Das Hauptaugenmerk in dieser Arbeit ist daher auf das effiziente Lösen von Constraintsystemen gerichtet. Darüberhinaus wird ein kleiner Einblick in die Verarbeitung von Constraint-Hierarchien und in die constraintbasierte Logikprogrammierung gegeben.

1.2. Einführendes Beispiel

Constraintsysteme werden in vielen praktischen Anwendungsbereichen eingesetzt. Leider kann im Rahmen dieser Ausarbeitung nicht näher auf reale Beispiele eingegangen werden. Um aber den anschaulichen Charakter nicht aus den Augen zu verlieren, soll folgende Aufgabe als Beispiel für ein kleines Constraintsystem gelten.

Abbildung 1: Karte von Australien

Hierbei gilt es, die Landkarte von Australien einzufärben. Australien teilt sich in sieben Bundesstaaten auf. Jeder Bundesstaat soll mit einer von drei zur Auswahl stehenden Farben belegt werden. Hierbei ist zu beachten, dass zwei benachbarte Bundesstaaten nicht mit derselben Farbe eingefärbt werden dürfen.

Bevor auf einen möglichen Lösungsweg eingegangen wird, soll zunächst analysiert werden, welche Elemente durch die Aufgabenstellung gegeben sind. Hierbei lassen sich drei wichtige Gruppen herausfiltern: Variablen, Wertebereiche der Variablen und Bedingungen der Variablen untereinander.

Variablen:	WA, NT, SA, Q, NSW, V, T
Wertebereich:	{rot, grün, blau}
Einschränkungen:	$WA \neq SA \wedge WA \neq NT \wedge NT \neq SA \wedge NT \neq Q \wedge$ $Q \neq SA \wedge Q \neq NSW \wedge NSW \neq SA \wedge NSW \neq V \wedge V \neq SA$

Dieses Schema von Variablen, Wertebereichen und auswertbare Relationen auf den Variablen wird im allgemeinen als Constraintsystem bezeichnet.

2. Grundlegende Begriffe

Bevor auf die eigentlichen Problemstellungen von Constraintsystemen eingegangen werden kann, ist es notwendig, wichtige Begriffe einzuführen.
Ein einzelnes Constraint kann als eine einzelne Bedingung aufgefasst werden. Hierbei werden beliebig viele Variablen mittels Operatoren in Beziehung zueinander gesetzt. Jede Variable hat einen eigenen Wertebereich (Domain). Die einfache Relation $x=y+4$ mit $x, y \in \mathbb{R}$ kann zum Beispiel als ein einzelnes Constraint mit den Variablen x und y betrachtet werden.
Unter der Stelligkeit eines Constraints versteht man die Anzahl der Variablen in einer Bedingung. Bei einer einzigen Variablen, z.B. $x=4$, spricht man von einem unären Constraint, bei zwei Variablen ($x=y+4$) von einem binären.
Unter einem Constraintnetz, bzw. Constraintsystem, versteht man eine Menge von einzelnen Constraints. Die schon erwähnten Beispiele von einzelnen Constraints können als Constraintsysteme mit nur einer einzigen Bedingung aufgefasst werden. Ein etwas komplexeres Netz ist z.B. durch folgende Definition gegeben: $(x<y) \wedge (x<z) \wedge (y<z) \wedge x, y, z \in \{1,2,3\}$. Dieses System enthält die Constraints $x<y$, $x<z$ und $y<z$, sowie eine Angabe zum Wertebereich der Variablen.
Constraintnetzen kann ebenfalls unter bestimmten Gegebenheiten eine Stelligkeit zugeordnet werden. So versteht man unter einem binären Constraintnetz ein Constraintsystem, dass ausschließlich aus binären und unären Constraints besteht. Es kann gezeigt werden, dass man alle Constraints mit einer Stelligkeit höher als zwei auf binäre Constraints runterbrechen kann, wenn die Wertemengen aller Variablen endlich sind. Ein binäres Constraintsystem kann bildlich durch einen ungerichteten Graphen dargestellt werden. Hierbei stehen die Knoten für einzelne Variablen und die Verbindungen der Knoten repräsentieren die einzelnen Constraints. Das kleine Constraintnetz aus dem einführenden Beispiel kann wegen der vorhandenen Stelligkeit von zwei bei allen Bedingungen auch als Graph dargestellt werden.

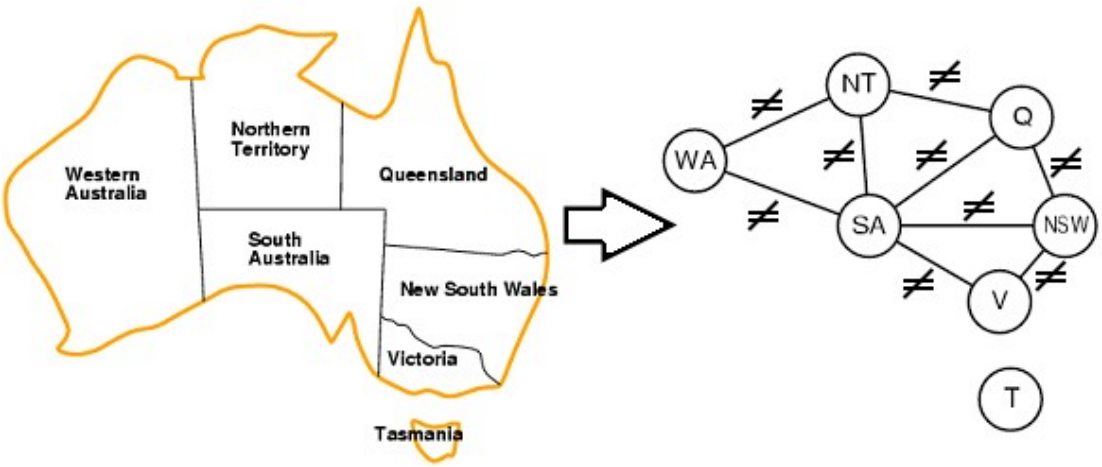

Abbildung 2: Constraintgraph Australien

Existiert für ein Constraint eine Variablenbelegung, so dass kein Widerspruch auftritt, dann spricht man davon, dass diese Belegung das Constraint erfüllt. Kann eine Variablenbelegung gefunden werden, die alle Constraints des Systems erfüllt, dann liegt eine Lösung für das Constraintnetz vor. Die Frage, ob und welche Lösungen für ein Constraintsystem gefunden werden können, wird als Constraint-Satisfaction-Problem (CSP) bezeichnet.

3. Lösen von Constraint-Satisfaction-Problemen (CSP)

3.1. Grundlagen von CSPs

Das Hauptaugenmerk bei der Behandlung von Constraintsystemen besteht in ihrer Lösbarkeit. Hierbei treten zwei Problemstellungen auf:

1. Existiert eine Lösung für das Constraintsystem?
2. Ausgabe einer Lösung, falls überhaupt eine existiert.

Beide Fragen entsprechen einander. Wenn nachgewiesen wurde, dass ein Constraint-system eine Lösung besitzt, dann ist auch meistens bekannt, wie die Lösung lautet.

3.2. Gewünschte Eigenschaften von Solvern

Ein Programm, das ein Constraintsystem entgegen nimmt und entscheidet, ob dieses Lösbar ist, wird als Solver bezeichnet. Liegt eine Lösung vor, so soll auch die entsprechende Variablenbelegung angegeben werden. Diese Programme können verschiedene Eigenschaften bezüglich ihres Verhaltens haben.

Eine grobe Unterteilung von Solvern kann vorgenommen werden, indem man sie in vollständige und unvollständige einteilt. Vollständige Solver können immer entscheiden, ob ein Constraintsystem lösbar ist. Bei unvollständigen Solvern kann es vorkommen, dass keine Aussage getroffen werden kann. In diesem Fall liefert der Solver das Ergebnis „unkown“. Obwohl unvollständige Solver auf den ersten Blick nicht sehr praxistauglich erscheinen, wird doch des öfteren auf solche Algorithmen zurückgegriffen. Der Grund hierfür ist, dass vollständige Solver-Algorithmen meistens ein sehr schlechtes Laufzeitverhalten haben, wohingegen unvollständige in der Regel die Berechnung schneller durchführen können, wenn sie für einen bestimmten Typ von Constraintsystemen optimiert sind.

Unvollständige Solver sollten in ihrem Lösungsverhalten aber bestimmte Richt-

linien einhalten, damit sie in der Praxis eingesetzt werden können.

1. die Lösungssaussage darf nicht abhängig von den verwendeten Variablennamen sein (variable name independent). Sollte eine Umbenennung stattfinden, so darf z.B. nicht ein vorher entscheidbares Problem mit „unkown“ gekennzeichnet werden.
2. die Reihenfolge der einzelnen Constraints darf keine Rolle spielen. Wichtig für die Lösungsaussage ist nur die Menge der Einzel-Constraints (set based).
3. Ein Solver sollte die konjunktive Verknüpfung von einzelnen Constraints beachten (monotonic). Ist z.B. das Constraint C_1 vom Solver als unlösbar deklariert worden, so sollte auch bei dem Constraintsystem $C_1 \wedge C_2$ auf unlösbar entschieden werden und nicht auf „unkown“.

Ein unvollständiger Solver, der diese Eigenschaften besitzt, wird als „well-behaved“ bezeichnet. Vollständige Solver sind immer „well-behaved“.

3.3. unendliche Wertemengen

Das Lösen von Constraintsystemen, die Variablen mit unendlichen Wertemengen besitzen, ist nicht immer möglich. Es ist nicht möglich, einen vollständigen Algorithmus zu implementieren, der ein beliebiges Constraintsystem lösen kann. Würde ein solcher Algorithmus gefunden werden, dann könnten alle Problemstellungen, die sich als CSP formulieren lassen auch gelöst werden. Dieses würde den Grundprinzipien der theoretischen Informatik widersprechen, wonach bestimmte Probleme (z.B. das Halteproblem) nicht lösbar sind.
Die existierenden Algorithmen können immer nur Constraintnetze mit einer bestimmten Form lösen, sie sind also problemspezifisch. So kann z.B. für lineare Gleichungssysteme der Gauß-Algorithmus benutzt werden. Auf andere Constraintsysteme, bei denen die Variablen nicht nur in der ersten Potenz vorkommen, ist dieser nicht anwendbar.
Bestimmte Autoren (Roman Bartak) sprechen bei der Lösung von Constraintsystemen mit Variablen mit unendlichen Wertebereichen nicht von „Constraint-

Satisfaction“, sondern von „Constraint-Solving“. In den Standardwerken zur KI (Russel/Norvig) findet diese Unterscheidung nicht statt.

3.4. endliche Wertemengen

Wie oben beschrieben kann ein allgemeiner Solver für Constraint-Systeme, bei denen die Wertemengen der Variablen nicht beschränkt sind, nicht gefunden werden. Anders verhält es sich hingegen bei Systemen mit endlichen Wertemengen. Beispiele für Variablen mit endlichen Wertemengen sind z.B. boolesche Variablen und Integer-Zahlen, bei denen eine obere und eine untere Schranke festgelegt wurde. Obwohl diese Einschränkung nicht sehr praxistauglich erscheinen mag, sind doch viele Probleme im täglichen Leben mit solchen Constraintsystemen modellierbar.
Der große Vorteil dieser Systeme gegenüber denen mit unendlichen Wertebereichen liegt im Vorhandensein eines allgemeinen vollständigen Solvers. Leider ist dieses Problem NP-vollständig, d.h. dass im schlimmsten Fall ein allgemeiner vollständiger Solver nur exponentielles Laufzeitverhalten zur Anzahl der Variablen besitzt.
Eine kleine Auswahl an möglichen Algorithmen, mit denen das CSP gelöst werden kann, wird in den folgenden Abschnitten gegeben.

3.5. CSP-Algorithmen

3.5.1. Konsistenz

Bevor mit der Lösung eines Constraintsystems begonnen wird, ist es ratsam, den Wertebereich der Variablen und die Werte zu verringern, die offensichtlich nicht zur Lösung des CSP beitragen können. Um herauszufinden, welche Variablenwerte vernachlässigt werden können, bedient man sich sogenannter Konsistenzbedingungen. Diese sind einer Ordnung unterstellt. Je höher der Grad einer Konsistenzbedingung, desto mehr Inkonsistenzen können erkannt

werden, desto höher wird aber auch der damit verbundene Rechenaufwand. Damit ein Constraintsystem eine Konsistenzstufe erreichen kann, müssen alle einzelnen Constraints der Konsistenzbedingung genügen.

Die Knotenkonsistenz ist die einfachste Bedingung. Sie untersucht ausschließlich unäre Constraints und entfernt alle Werte, die diesen Einschränkungen nicht genügen. Bei vielen Systemen wird auf unäre Constraints verzichtet, indem diese Bedingungen gleich in die Wertemengen eingearbeitet werden. In so einem Fall ist das Constraint-Netz immer knotenkonsistent.

Das System $x<4 \wedge y>2 \wedge x<y$ mit $x, y \in \{1,2,3,4\}$ ist nicht knotenkonsistent, weil bei beiden Variablen Werte vorstellbar sind, die nicht die unären Constraints erfüllen. Nach Entfernung aller unzulässigen Werte erhält man das folgende System: $x<4 \wedge y>2 \wedge x<y$ mit $x \in \{1,2,3\} \wedge y \in \{3,4\}$. Die unären Constraints können nach diesen Schritt weggelassen werden, weil sichergestellt ist, dass sie immer erfüllt sind. Daher kann das System auch als $x<y$ mit $x \in \{1,2,3\} \wedge y \in \{3,4\}$ geschrieben werden.

Den nächsten Konsistenzgrad bezeichnet man als Kantenkonsistenz. Hier werden alle binären Constraints untersucht. Die Konsistenzbedingung ist für ein Constraint erfüllt, wenn für jeden Wert aus dem Wertebereich der einen Variable auch ein Wert aus der Wertemenge der anderen Variable gefunden werden kann, für die das binäre Constraint gilt. Alle Werte, für die kein entsprechender Partner existiert, werden aus den Wertemengen entfernt.

Im Gegensatz zur Knotenkonsistenz kann die einmalige Anwendung der Kantenkonsistenzregel weitere Inkompatibilitäten erzeugen. Daher ist es notwendig, die Kantenkonsistenzregel solange anzuwenden, bis bei einem Durchgang bei keiner Variable ein Wert entfernt wurde.

Das Constraintsystem $x<y \wedge x<z \wedge y<z$ mit $x, y, z \in \{1,2,3,4\}$ ist nicht kantenkonsistent. Bei der Betrachtung des Einzel-Constraint $x<y$ kann nicht für jeden Wert von x und y ein entsprechender Partner gefunden werden. So ist das Constraint für $x=4$ oder $y=1$ nicht erfüllbar. Diese Werte müssen aus den Wertemengen entfernt werden. Das so entstandene System mit den Wertebereichen $x \in \{1,2,3\} \wedge y \in \{2,3,4\} \wedge z \in \{1,2,3,4\}$ ist jedoch immer noch nicht kantenkonsistent. Daher müssen weitere Durchläufe ausgeführt werden. Nach Abschluss des Verfahrens sind die Wertebereiche auf folgende Mengen reduziert: $x \in \{1,2\} \wedge y \in \{2,3\} \wedge z \in \{3,4\}$.

Die Tatsache, dass ein Constraint-System sowohl knoten- als auch kantenkonsistent ist, impliziert nicht, dass für dieses System zwangsweise eine Lösung existieren muss. Durch Kantenkonsistenz wird immer nur ein Constraint zur Zeit untersucht, daher werden Inkompatibilitäten, die bei der Betrachtung von mehreren binären Konstraints auftreten, nicht entdeckt.

Das System $x \neq y \wedge x \neq z \wedge y \neq z$ mit $x, y, z \in \{1,2\}$ ist kantenkonsistent, weil bei der Betrachtung der einzelnen Constraints für jede Variablenbelegung immer ein Wert der anderen Variablen gefunden werden kann. In seiner Gesamtheit besitzt das Netz aber keine Lösung.

Um Constraints mit höherer Stelligkeit konsistent zu machen, müssen auch höhere Konsistenzgrade erreicht werden. Dieses Verfahren wird häufig als k-Konsistenz bezeichnet, wobei k die Stelligkeit der Constraints angibt, die untersucht werden sollen. Hierbei steigt allerdings der Rechenaufwand soweit an, dass ein vollständiger Lösungsalgorithmus meistens bessere und sichere Resultate erzielt.

In der Praxis werden aber dennoch einfache, unvollständige CSP-Solver implementiert, die ausschließlich auf Knoten- und Kantenkonsistenz beruhen. Hierbei kann ein CSP als unlösbar eingestuft werden, wenn entweder die Wertemenge einer Variablen leer wird, oder nach Herstellung der Konsistenz alle Variablen nur einen Wert annehmen, aber dennoch einige Constraints verletzt werden. Eine Aussage über eine Lösung kann nur getroffen werden, wenn alle Variablen ebenfalls fest bestimmt sind und alle Constraints erfüllt sind. Sollten jedoch mehrere Werte für eine Variable zulässig sein, so kann die Antwort auf das CSP nur „unkown“ lauten.

3.5.2. Generate and Test

In den hier behandelten Constraintnetzen kann jede Variable nur eine endliche Menge an Werten annehmen. Es existiert daher nur eine endliche Menge an Belegungen für das Constraintsystem. Folglich ist eine Generierung aller möglichen Belegungen der Variablen möglich. Jede dieser Belegungen kann dann darauf untersucht werden, ob sie eine Lösung des Constraint-Systems darstellt.

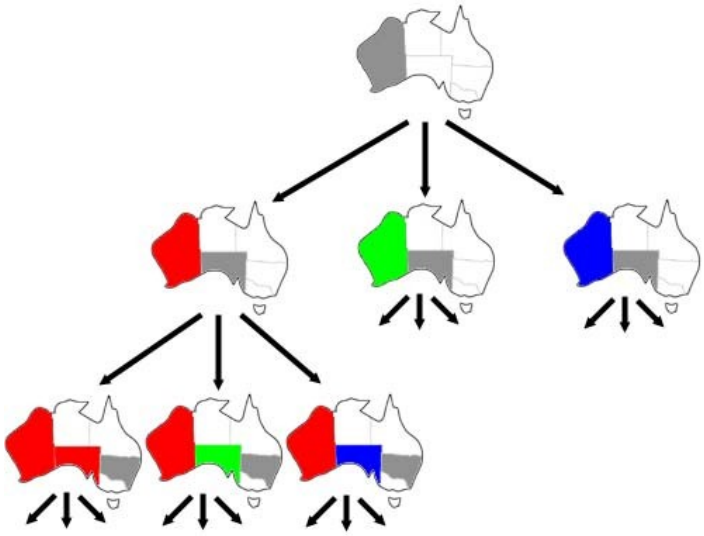

Abbildung 3: Generate and Test - Suchbaum

Zur besseren Visualisierung der Vorgehensweise eignet sich die Abbildung aller möglichen Belegungen in einer Baumstruktur. Als Beispiel ist an dieser Stelle das Australien-Beispiel angeführt.

Es zeigt sich, dass die Menge der Kombinationsmöglichkeiten identisch ist mit dem Kreuzprodukt der Anzahl der Elemente in den Wertebereichen der Variablen. Schon bei diesem kleinen Beispiel existieren fast 2200 verschiedene Belegungen. Die Generierung von neuen Belegungen kann abgebrochen werden, wenn eine Lösung gefunden wurde. Sollte das System aber keine Lösung besitzen, so müssen alle Kombinationen getestet werden.

Das hier beschriebene Vorgehen kann kaum als Algorithmus für die Lösung eines CSP herhalten. Es soll aber verdeutlichen, warum es für Constraintsysteme, bei denen die Variablen einen endlichen Wertebereich besitzen, immer einen Lösungsmechanismus gibt. Natürlich ist dieser Algorithmus in der Praxis nicht anwendbar. Auf Basis dieses Lösungsansatzes können aber Algorithmen gefunden werden, die ein besseres Laufzeitverhalten besitzen. Leider besitzen auch diese Algorithmen im Worst-Case ebenfalls ein exponentielles Laufzeitverhalten bezogen auf die Anzahl der Variablen.

3.5.3. Backtracking

3.5.3.1. Chronologisches Backtracking

Der bekannteste Algorithmus zur Lösung von CSPs ist das Backtracking- oder auch Rücksetz-Verfahren. Im Gegensatz zur Generate-and-Test-Methode wird versucht, die Anzahl der zu testenden Belegungen klein zu halten, indem inkonsistente Belegungen möglichst früh erkannt werden.
Bei diesem Verfahren wird nacheinander jeder Variablen ein Wert zugeordnet. Zwischen den Wertzuweisungen wird geprüft, ob die aktuelle Belegung Inkonsistenzen hervorruft, d.h. ob einige Constraints verletzt werden. Ist dies der Fall, so wird der Wert der zuletzt belegten Variablen verworfen und ein anderer zugewiesen. Das Verfahren endet, wenn alle Variablen mit Werten belegt sind und keine Inkonsistenten auftreten. In diesem Fall ist eine Lösung für das CSP gefunden. Sollte der Fall eintreten, dass keine weiteren Belegungsmöglichkeiten mehr gegeben sind, dann bricht dieser Algorithmus ebenfalls ab, wobei sichergestellt ist, dass das vorliegende Constraintsystem unlösbar ist.

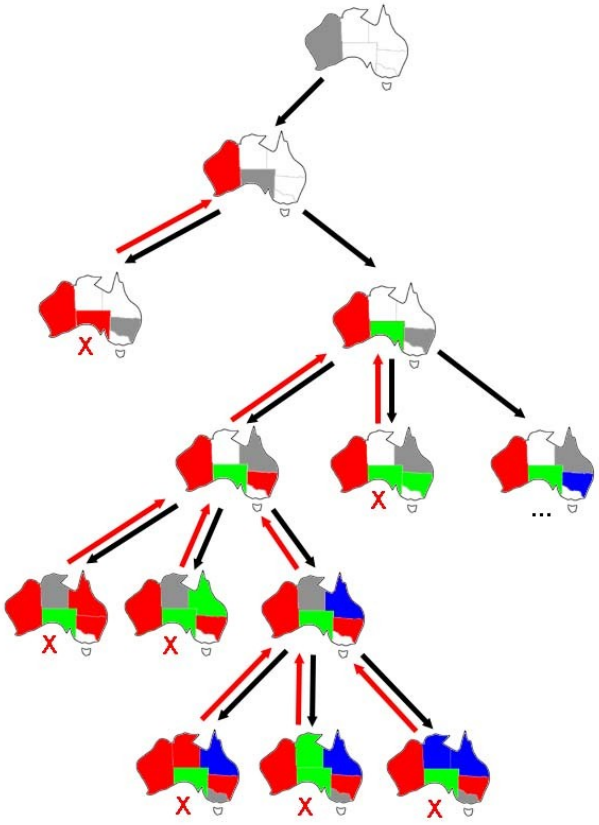

Abbildung 4: Backtracking - Suchbaum

Wie man erkennt, werden nicht alle möglichen Pfade des Baumes durchlaufen. Allerdings müssen auch bei diesem kleinen Beispiel relativ viele Rücksetzschritte (rote Pfeile) gemacht werden, bis eine Lösung gefunden ist. Eine Optimierung dieses Algorithmus hat daher das primäre Ziel, die Anzahl der Rücksetzungen gering zu halten und möglichst auf direktem Weg die Lösung zu erreichen. In der Beschreibung des Backtracking-Algorithmus werden wichtige Fragen nicht beantwortet, die aber Anstöße für Verbesserungen liefern:

1. Welche Variable soll als nächstes instanziiert werden und mit welchen Wert soll dies geschehen?
2. Welche Folgen können aus der aktuellen Variablenbelegung auf zukünftige Wertzuweisungen geschlossen werden?
3. Wenn sich herausstellt, dass ein Suchpfad inkonsistent wird, kann dann dieser Fehler in zukünftigen Suchschritten vermieden werden?

3.5.3.2. Heuristiken

Zur Beantwortung der ersten Frage werden spezielle Heuristiken herangezogen, mit denen eine bessere Instanziierungsreihenfolge gefunden werden soll. Es soll erreicht werden, dass möglichst früh Inkonsistenten erkannt werden und damit die Anzahl der zu testenden Kandidaten klein gehalten wird. Hierbei wird zwischen zwei unterschiedlichen Typen von Heuristiken unterschieden. Variablen-Ordnungsheuristiken legen fest, welche Variable als nächstes instanziiert werden soll, wobei Werte-Ordnungsheuristiken eine Aussage über den nächsten Wert der Variablen machen. Da beide Verfahren von einander unabhängig sind und sich gegenseitig ergänzen, können sie in der Praxis gemeinsam verwendet werden.
Eine generelle Aussage, welche Heuristik die besten Resultate erzielt, kann nicht getroffen werden. Meist hängt dies von den zu lösenden Problem ab. Die folgende Auflistung gibt eine kleine Auswahl über die gängigsten Ordnungsheuristiken wieder:

Variablen-Ordnungsheuristiken:

Minimum remaining values (MRV):

Die Variable mit den wenigsten zulässigen Werte ist auszuwählen

Maximum-Degree-Heuristic:

Die Variable ist auszuwählen, die am meisten in Constraints vorkommt, in denen Variablen noch nicht festgelegt sind.

Werte-Ordnungsheuristiken:

Least-constraining value:

Der Wert mit den wenigsten folgenden Einschränkungen ist zu nehmen.

3.5.3.3. Forward-Checking

Neben der Verwendung von Heuristiken kann der Suchraum beim Backtracking-Verfahren weiter eingeschränkt werden. Eine effiziente Möglichkeit ist, dass bei jeder Instanziierung einer Variablen alle Werte aus den Wertemengen anderer Variablen entfernt werden, die im Widerspruch zu der getroffenen Instanziierung stehen. Hierbei werden aber nur die Variablen auf Inkonsistenzen untersucht, die mit der aktuell behandelten durch ein Constraint verbunden sind. Dadurch entfällt die Untersuchung aller Werte, die von vornherein eine Inkompatibilität zur aktuellen Belegung aufweisen. Dieses Vorgehen wird als Forward-Checking bezeichnet. Das folgende Beispiel verdeutlicht die Arbeitsweise dieses Verfahrens.

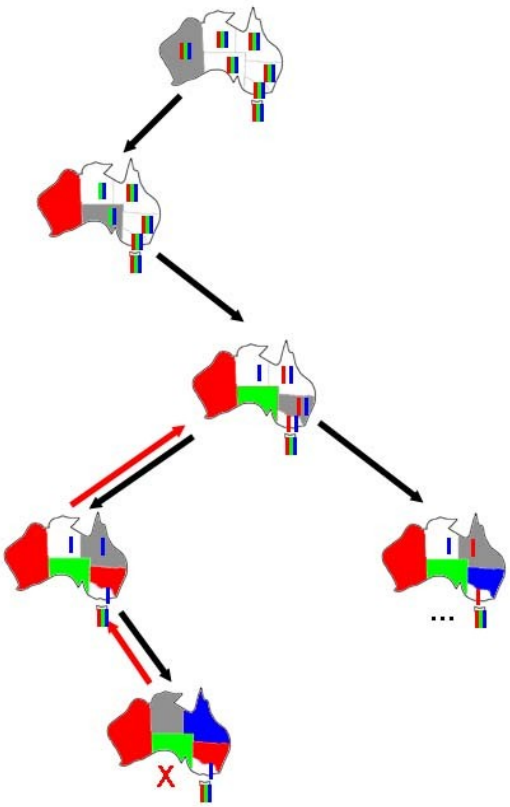

Abbildung 5: Forward-Checking - Suchbaum

Im Vergleich zum Backtracking-Suchbaum ist hier die Anzahl der Backtracking-Schritte weiter reduziert worden. Die farbigen Balken in den Bundesstaaten verdeutlichen, welche Farben für die Bundesstaaten noch möglich sind und welche im Zuge des Forward-Checks aus den Wertebereichen entfernt wurden. Forward-Checking wird oft zusammen mit der MRV-Heuristik angewandt, da durch Forward-Checking schon alle unzulässigen Werte heraus gefiltert werden und daher nur noch die Variable ausgewählt werden muss, die den kleinsten verbleibenden Wertebereich besitzt.

3.5.3.4. MAC-Algorithmus

Obwohl mit der Methode des Forward-Checking viele Inkonsistenzen schon vor dem Test ausgeschlossen werden können, bleiben doch einige unentdeckt.
Um herauszufinden, welche Auswirkung die Belegung einer Variable auf die Wertebereiche der anderen hat, greift man auf die oben beschriebenen Methoden zur Konsistenzherstellung zurück. Meist genügt in der Praxis die

Anwendung der Kanten-Konsistenz-Algorithmen. Höhere Konsistenzgrade bedingen einen höheren Rechenaufwand, so dass meistens keine Verbesserung der Laufzeit erreicht werden kann.
Die Herstellung der Kanten-Konsistenz kann nur einmal vor dem Starten des Backtracking-Verfahrens stattfinden, aber auch nach jedem Instanziierungsschritt. Letztere Variante wird als MAC-Algorithmus (Maintaining Arc Consistency) bezeichnet.
Der folgende Suchbaum verdeutlicht die Arbeitsweise des MAC-Algorithmus. Die Backtracking-Schritte wurden vermieden, allerdings bedeutet die Herstellung der Kantenkonsistenz nach jeder Instanziierung auch einen höheren Rechenaufwand.

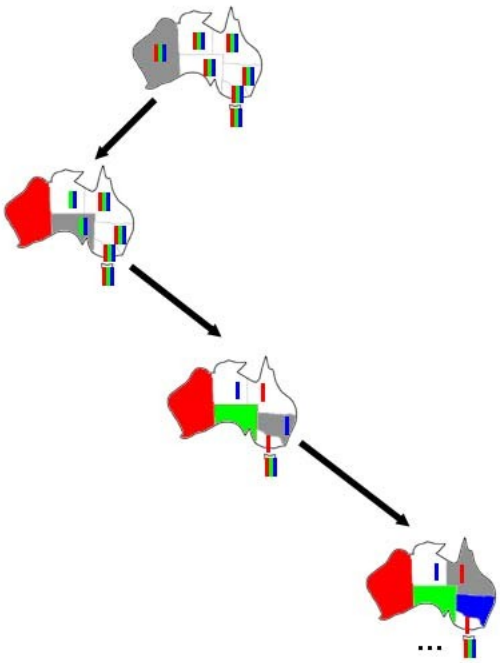

Abbildung 6: MAC-Algorithmus - Suchbaum

3.5.4. Min-Conflicts

Neben den Backtracking-Algorithmus gibt es eine Reihe weiterer Verfahren zur Lösung von CSPs. Eine in der Praxis häufig eingesetzte Methode ist die der Min-Conflicts-Heuristik. Im Gegensatz zum Backtracking handelt es sich nicht um eine systematische Suche, sondern um eine stochastische. Hierzu wird als Startpunkt eine zufällige Belegung der Variablen erzeugt. Ist diese keine

Lösung des CSP –wovon in der Regel auszugehen ist–, wird eine konfliktverursachende Variable ausgewählt und versucht, einen neuen Wert für diese zu finden, wobei aber weniger Constraints verletzt werden sollen als zuvor.

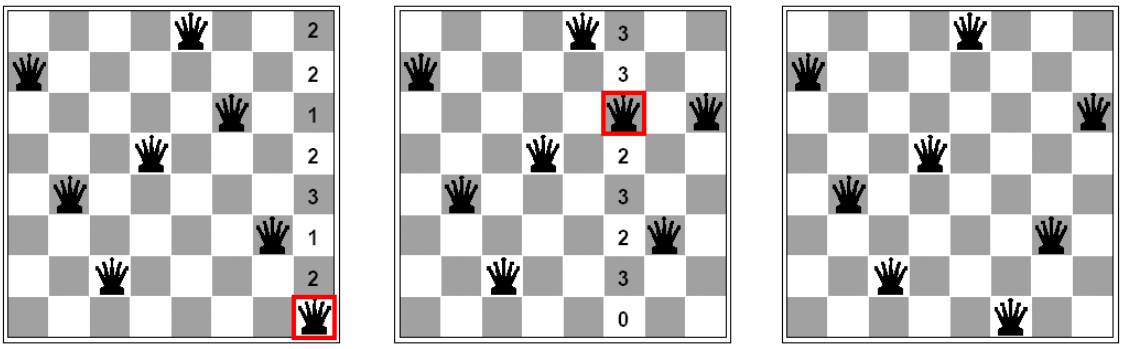

Abbildung 7: Min-Conflicts - Beispiel

Eine effektive Anwendung dieses Algorithmus ist die Lösung des 8-Damen-Problems. Hierbei sind acht Damen so auf einen Schachbrett aufzustellen, dass keine Dame in der Lage ist, eine andere Dame zu schlagen. Hierbei ist zu beachten, dass jeweils zwei Damen nicht in der selben Zeile, Spalte oder Diagonalen stehen dürfen.

Zunächst wird eine willkürliche Verteilung der Damen auf dem Schachbrett gewählt. Anschließend wird eine Dame ausgewählt, die im Konflikt mit einer anderen steht. Es wird berechnet, auf welchen Feld die wenigsten Konflikte mit anderen Damen auftreten und die aktuelle Figur wird auf dieses Feld gestellt. Anschließend wird dieses Verfahren mit anderen konfliktverursachenden Figuren solange wiederholt, bis eine Lösung gefunden wurde.

Wie bei vielen anderen stochastischen Methoden besteht auch hier die Gefahr, dass das System in einen lokalen Minimum stecken bleibt. Das bedeutet, dass durch die oben dargestellte Vorgehensweise keine weiteren Verbesserungen erzielt werden können, eine Lösung für das CSP aber noch nicht gefunden wurde. Für diesen Fall muss von der normalen Methode abgewichen werden und versucht werden, die aktuelle Variablenbelegung anderweitig in einen konsistenteren Zustand zu überführen (Random-Walk). Um die Gefahr eines „Im-Kreis-Laufens“ zu verhindern, können mit Hilfe einer Tabu-Liste die letzten Belegungsschritte protokolliert werden und entsprechende Gegenmaßnahmen getroffen werden.

Dieser Algorithmus wird manchmal auch als „heuristisches Reparieren“ bezeichnet. Damit soll angedeutet werden, dass meistens eine Lösung für ein

CSP gefunden wurde, sich aber das Constraint-System geringfügig verändert hat. Anstatt in einem solchen Fall wieder Backtracking anzuwenden und damit zu riskieren, eine komplett andere Lösung zu bekommen, wird durch dieses Verfahren versucht, die alte Lösung so gering wie möglich zu verändern.

3.5.5. Struktur von Constraint-Graphen

Wenn – wie in der Einleitung besprochen – das Constraintsystem als Graph darstellt wird, gewinnt man leicht einen Eindruck über die Komplexität des zu lösenden CSP.
Dabei ist man bestrebt, das Gesamtproblem in Teilprobleme zu zerlegen. Hierdurch kann in der Regel eine Verbesserung des Laufzeitverhaltens erreicht werden. Wenn es im Constraint-Graph zwei Teile gibt, die nicht miteinander verbunden sind, so handelt es sich dabei um zwei unabhängige Teilprobleme. Diese können getrennt voneinander gelöst werden. Dieser Fall tritt aber in der Praxis recht selten auf.

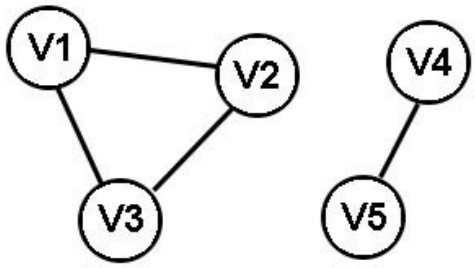

Abbildung 8: Constraintgraph - zwei Teilprobleme

Als einfach zu lösende Constraintnetze haben sich solche herausgestellt, bei denen die Knoten im Graph in einer Baumstruktur zueinander stehen. Bei einer solchen Konstellation ist das Laufzeitverhalten linear zur Menge der Variablen. Ausgehend von den Blättern des Baumes wird Kantenkonsistenz zu den jeweiligen Vaterelementen hergestellt. Im letzten Schritt werden die Variablen von der Wurzel des Baumes ausgehend mit Werten belegt, wobei hierbei wieder auf Inkonsistenzen zu achten ist. Ein zurücknehmen von Variablenbelegungen wie beim Backtracking ist aber durch das Herstellen der Kantenkonsistenz in

diesem Fall nicht nötig.

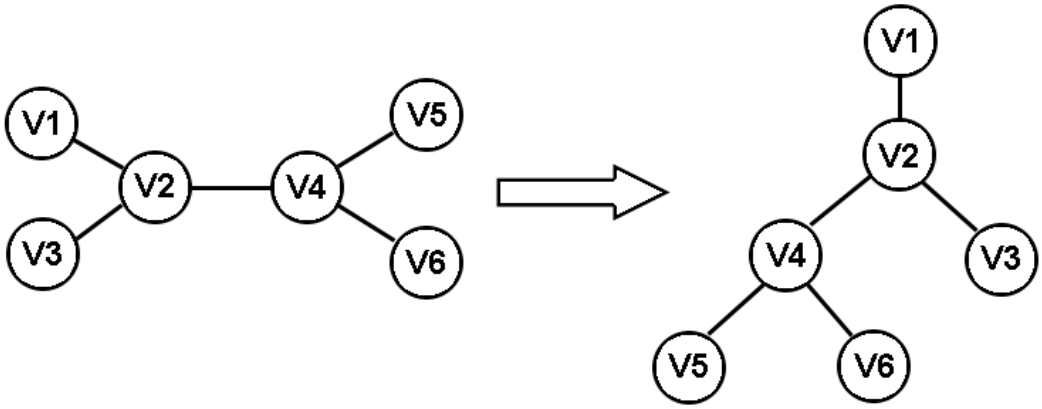

Abbildung 9: Constraintgraph - Baumstruktur

Um auch Constraintsysteme mit dieser Methode lösen zu können, in denen die Variablen im Constraint-Graph nicht in einer Baumstruktur miteinander verbunden sind, ist eine Transformation des Graphen in einem Baum erforderlich.

Dabei kann zum einen versucht werden, durch gezieltes Entfernen von Knoten aus dem Graphen eine Baumstruktur zu gewinnen. Hierzu werden einige Variablen mit Werten belegt, so dass der Graph um diese Knoten verringert wird. Natürlich kann die Wahl der Vorbelegung ungeschickt gewählt sein, so dass für den verbleibenden Baum keine Lösung existiert. Tritt dieser Fall ein, so muss das Verfahren mit einer anderen Vorbelegung wiederholt werden. Diese Technik – auch bekannt als „cutset-conditioning“ – bietet sich an, wenn der Constraint-Graph schon eine baumähnliche Struktur besitzt. Als Verdeutlichung kann hier wiederum das Australien-Beispiel betrachtet werden.

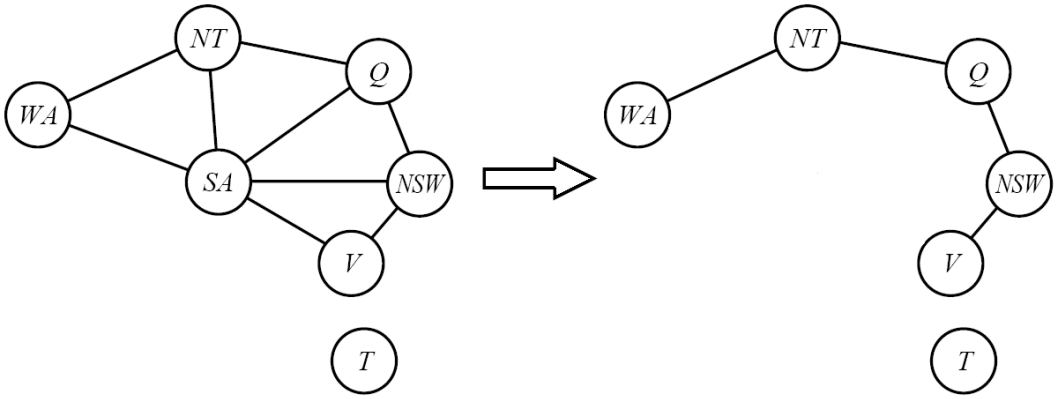

Abbildung 10: cutset conditioning - Australien

Eine andere Möglichkeit zur Transformation eines Constraint-Graphen in eine

Baumstruktur besteht darin, einige Knoten zu Unterproblemen zusammenzufassen. Diese neu entstandenen Unterprobleme sind dann miteinander in einer Baumstruktur organisiert. Die Lösungen für diese Unterprobleme können dann separat gefunden werden. Die Gesamtlösung wird dann auf Basis des Algorithmus für Baumstrukturen bestimmt. Bei der Zusammenfassung der Knoten sind jedoch einige Regeln zu beachten:

1. Jede Variable aus dem ursprünglichen Problem muss in mindestens einem Unterproblem auftauchen.
2. Wenn zwei Variablen im ursprünglichen Problem durch ein Constraint verbunden sind, so müssen sie auch in mindestens einem Unterproblem durch dieses verbunden sein.
3. Wenn eine Variable in zwei Unterproblemen im neuen Graph auftaucht, dann muss sie auch in jedem Unterproblem auftreten, das auf dem Weg zwischen den beiden Unterproblemen liegt.

In der Literatur wird dieses Vorgehen auch als „tree-decomposition“ bezeichnet.

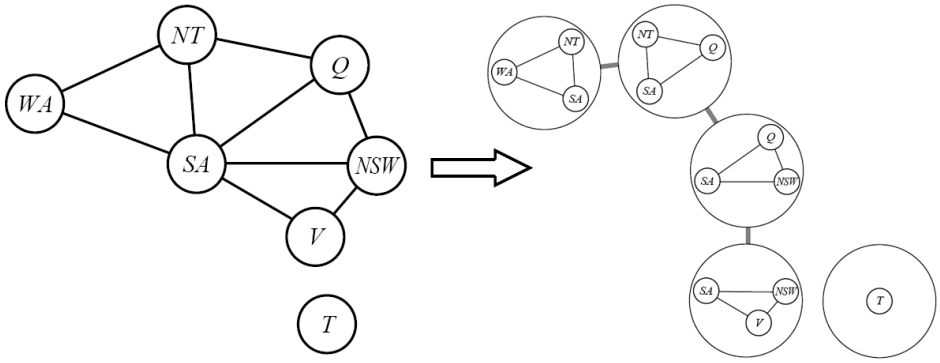

Abbildung 11: tree decomposition - Australien

4. Constraint-Hierarchien

4.1. „harte“ / „weiche“ Constraints

In den bisher behandelten Constraintsystemen sind alle Constraints gleich gewichtet. Wenn eine Bedingung verletzt wird, dann ist das gesamte System nicht erfüllt. In vielen Anwendungen – besonders bei Zeitplanungen – ist diese Vorgehensweise nicht immer erwünscht. So kann es bei der Erstellung eines Vorlesungsverzeichnisses für eine Hochschule verschiedene Randbedingungen geben, die leider nicht alle gleichzeitig eingehalten werden können. In diesem Fall ist eine Gewichtung der Constraints angebracht. Hierzu wird das Constraintsystem in mehrere Systeme aufgeteilt, wobei jedem Untersystem eine Hierarchiestufe bzw. ein Level zugeordnet wird, die untereinander nummeriert sind. Die Stufe 0 beinhaltet alle Constraints, die unbedingt erfüllt sein müssen, und auf die nicht verzichtet werden kann. Die anderen Stufen beinhalten optionale Constraints, wobei die Bedingungen eines Levels mit einer kleineren Ordnung wichtiger („härter“) sind als eine mit einer höheren Ordnung. Darüberhinaus wird eine Fehlerfunktion für jedes Constraint definiert. Ein Fehlerwert von 0 bedeutet, dass das Constraint erfüllt ist. Eine Lösung für die Constrainthierarchie versucht demnach möglichst viele Constraints zu erfüllen, wobei Constraints mit einer wichtigeren Priorität bevorzugt erfüllt werden. Um zwei Lösungen für das Hierarchiesystem vergleichen zu können, ist der Einsatz von Komparatoren unumgänglich. Hierbei haben sich einige Vergleichsmethoden in der Praxis als äußerst nützlich erwiesen.

4.2. Komparatoren

Im Rahmen dieser Ausarbeitung soll kurz auf die Locally- und Globally-Komparatoren („locally-better“ bzw. „globally-better“) eingegangen werden. Die Locally-Vergleichsmethoden betrachten immer nur die Fehlerwerte eines einzelnen Constraints und treffen auf Basis dieses Kriteriums eine Entscheidung.

Lösung 1			Lösung 2		
Constraint	**Level**	**Fehler**	**Constraint**	**Level**	**Fehler**
C1	1	0	C1	1	0
C2	1	0,5	C2	1	0,5
C3	2	0,8	C3	2	0,8
C4	2	0,4	C4	2	0,7
C5	3	0,8	C5	3	0,2

Tabelle 1: locally-better - Beispiel 1

An diesem Beispiel erkennt man die Arbeitsweise des Locally-Komparators. Es wird der Fehler jedes einzelnen Constraints betrachtet und mit der anderen Lösung verglichen. In diesem Beispiel ist Lösung 1 besser als Lösung 2. Beide erfüllen die Constraints C1, C2 und C3 gleich gut, aber C4 wird von der Lösung 1 besser erfüllt. Das Constraint C5 befindet sich in einer niedrigeren Prioritätsstufe, daher ist seine Erfüllung nicht so wichtig als die von C4.
Es kann aber vorkommen, dass der Locally-Komparator nicht immer zwei Lösungen miteinander vergleichen kann.

Lösung 1			Lösung 2		
Constraint	**Level**	**Fehler**	**Constraint**	**Level**	**Fehler**
C1	1	0	C1	1	0,4
C2	1	0,8	C2	1	0,5
C3	2	0,5	C3	2	0,2
C4	2	0,5	C4	2	0,6

Tabelle 2: locally-better - Beispiel 2

Diese Lösungen lassen sich mit einem Locally-Komparator nicht vergleichen, da in einer Hierarchiestufe bei zwei verschiedenen Constraints sowohl die eine als auch die andere Lösung einen niedrigeren Fehlerwert aufweist. Abhilfe verschaffen hier die Globally-Komparatoren, die den Gesamtfehler in einer Stufe ausrechnen und dann levelweise die Lösungen miteinander vergleichen.

Lösung 1				*Lösung 2*			
Constraint	***Level***	***Fehler***	***Summe***	***Constraint***	***Level***	***Fehler***	***Summe***
C1	1	0	0,8	C1	1	0,4	0,9
C2	1	0,8		C2	1	0,5	
C3	2	0,5	1	C3	2	0,2	0,8
C4	2	0,5		C4	2	0,6	

Tabelle 3: globally-better - Beispiel

Im obigen Beispiel kann nun ein Vergleich der Lösungen vorgenommen werden. Mit diesem Komparator ist die Lösung 1 die bessere, da der Gesamtfehler im Level 1 niedriger ist als bei Lösung 2.
Der Namenszusatz „predicate“ (locally-predicate-better bzw. globally-predicate-better) weist darauf hin, dass die Fehlerfunktion trivial ist. Ist die Bedingung erfüllt, so beträgt der Fehler 0, sonst 1.

4.3. Incremental Hierarchical Constraint Solver (IHCS)

Das Finden einer möglichst optimalen Lösung für eine Constraint-Hierarchie kann nicht mit den einfachen CSP-Algorithmen geschehen. In der Praxis werden meist Algorithmen verwendet, die für einen bestimmten Komparatortyp und einen bestimmten Wertebereich der Constraintvariablen zugeschnitten sind. An dieser Stelle soll kurz auf den Incremental-Hierarchical-Constraint-Solver (IHCS) eingegangen werden. Dieser Algorithmus ist in der Lage, eine Lösung für Constraint-Hierarchien mit Variablen auf endlichen Wertebereichen zu finden. Als Vergleichsmethode dient der globally-predicate-better-Komparator.
Dieser Algorithmus spaltet die Menge der Constraints in drei Teile auf, den Active-Store (AS), den Relaxed-Store (RS) und den Unexplored-Store (US). Der Active-Store enthält alle Constraints, die von der aktuellen Lösung des Constraints erfüllt werden, wohingegen im Relaxed-Store alle Bedingungen festgehalten werden, die verletzt werden. Der Unexplored-Store enthält alle Constraints, die noch nicht untersucht wurden. Als Darstellung der drei Mengen hat sich folgende Schreibweise etabliert: <AS•RS•US>.

Der Algorithmus geht davon aus, dass für eine gegebene Hierarchie eine Lösung gefunden wurde und sich alle Constraints entweder in AS oder in RS befinden (<AS•RS•{}>). Durch schrittweises Einfügen von neuen Constraints in US kann dann versucht werden, die Hierarchie zu erweitern. Hierzu werden zwei Regeln benötigt, die Forward-Rule und die Backward-Rule. Die Forward-Rule fügt einzelne Constraints aus US in AS ein. Sollten hierbei Konflikte entstehen, so wird die Backward-Rule ausgeführt. Diese verteilt unter Berücksichtigung des Komparators die Constraints aus AS und RS neu, so dass eine möglichst optimale Lösung gefunden werden kann. Das Ende dieses Prozesses ist erreicht, wenn die Menge US leer ist. Folgende Tabellen zeigen einen exemplarischen Ablauf dieses Verfahrens:

Hierarchie:

Name	*Constraint*	*Level*
c1	x + y = 15	1
c2	3 * x - y < 5	1
c3	x > y + 1	2
c4	x < 7	2

Tabelle 4: IHCS - Beispiel Hierarchie

Lösungsweg:

Aktion	*Konfiguration*	*D(X)*	*D(Y)*	*Regel*
add c1	{ }•{ }•{c1}	1..10	1..10	forward
	{c1 }•{ }•{ }	5..10	5..10	
add c2	{c1 }•{ }•{c2}	5..10	5..10	forward
	{c1,c2 }•{ }•{ }	-	-	backward
	{c1 }•{c2 }•{ }	5..10	5..10	
add c3	{c1 }•{c2 }•{c3}	5..10	5..10	forward
	{c1,c3 }•{c2 }•{ }	7..10	5..8	
add c4	{c1,c3 }•{c2 }•{c4}	7..10	5..8	forward
	{c1,c3,c4}•{c2 }•{ }	-	-	barkward
	{c1,c3 }•{c2,c4}•{ }	7..10	5..8	

Tabelle 5: IHCS - Beispiel Ablauf

5. Constraint Logic Programming (CLP)

Die softwarebasierte Implementierung von Constraintsolvern wurde in den letzten Jahren stark vorangetrieben. Hieraus sind viele Plattformen entstanden, die ein Modellieren und Lösen von Constraintsystemen ermöglichen. Dieses Konzept wird auch als Constraint Logic Programming (CLP) bezeichnet. Viele erfolgreiche Ansätze bauen auf den bestehenden Logik-Programmiersprachen wie z.B. PROLOG auf. Eines dieser CLP-Umgebungen ist das ECL^{i}PS^{e}-System. Es basiert auf PROLOG, erweitert jedoch den Sprachumfang um Bibliotheken, die es erlauben, Constraintsysteme mit endlichen Wertemengen zu lösen.

Ein einfaches Constraintprogramm besteht in der Regel aus Variablendefinitionen und anschließender Auflistung aller Constraints. Das eigentliche Lösen eines solchen Systems wird nicht vom Programmierer spezifiziert, sondern wird automatisch von der Laufzeitumgebung übernommen. Die eigentliche Arbeit des Programmierers besteht daher nicht mehr im Lösen eines Constraintsystems, sondern in dessen Modellierung. Ist diese abgeschlossen, so ist das Errechnen einer möglichen Lösung nur eine reine Formalität.

Das folgende kleine Programm löst das bekannte SEND-MORE-MONEY-Rätsel. Hierbei wird von folgender Rechenaufgabe ausgegangen:

```
  SEND
 +MORE
------
=MONEY
```

Abbildung 12: CLP - Beispiel

Die Buchstaben sind Zahlen zwischen 0 und 9 so zuzuordnen, dass die Rechnung stimmt. Jede Ziffer darf nur einmal an einem Buchstaben vergeben werden. Im ECL^{i}PS^{e}-System würde eine mögliche Modellierung so aussehen:

```
sendmore(Digits) :-
```

```
Digits = [S,E,N,D,M,O,R,Y],
Digits :: [0..9],
alldifferent(Digits),
S #\= 0,
M #\= 0,
              1000*S + 100*E + 10*N + D
           + 1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,
labeling(Digits).
```

Nach der Benennung der Routine folgt in den nächsten beiden Zeilen eine Definition der verwendeten Variablen. In diesem Fall wird ein Array mit acht Elementen angelegt, wobei jedes Element einen Wert zwischen 0 und 9 haben darf. In der vierten Zeile beginnt dann die Auflistung der Constraints:

$$
\begin{gathered}
S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y \wedge \\
S \neq 0 \wedge \\
M \neq 0 \wedge \\
1000 \cdot S + 100 \cdot E + 10 \cdot + D \\
1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\
= 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y
\end{gathered}
$$

Die Labeling-Anweisung sorgt für eine benutzerfreundlichere Ausgabe der Lösung. Der eigentliche Lösungsalgorithmus wird hier jedoch nicht beeinflusst.
Nach Ablauf des Programms in der ECL^{i}PS^e-Umgebung, erhält man folgende Lösung:

$$S = 9 \wedge E = 5 \wedge N = 6 \wedge D = 7 \wedge M = 1 \wedge O = 0 \wedge R = 8 \wedge Y = 2$$

Für dieses einfache Beispiel hält sich der resultierende Rechenaufwand in Grenzen. Jedoch zeigt es sich, dass bei größeren Problemen z.B. das 50-Damen-Problem schnell die Rechenzeit ansteigt. Vielleicht könnte der Einsatz einer besseren Heuristik den Rechenaufwand auf ein erträgliches Maß reduzieren.

6. Zusammenfassung

Constraints bzw. Constraintsysteme sind ein mächtiges Werkzeug um vielfältige Probleme zu modellieren und zu lösen. Obwohl das Konzept von Neben- und Randbedingungen weit verbreitet ist, wird doch meist selten explizit von Constraintssystemen gesprochen. Durch die formale Betrachtung dieses Themas lassen sich viele Problemstellungen leichter bearbeiten. Hierbei ist natürlich das Hauptaugenmerk auf die Lösungsalgorithmen gelegt. Es ist an dieser Stelle nochmals angemerkt, dass es den „perfekten“ Lösungsalgorithmus nicht gibt. Für viele Systeme muss abgewogen werden, welcher der effektivste ist.

Diese Arbeit kann natürlich nur eine Einführung in das Thema bieten. Besonders das Gebiet der CLP-Systeme und der hierarchischen Constraintsysteme liefert noch viele Möglichkeiten für weitere Betrachtungen.

Literaturverzeichnis

Günter Görz / Claus-Rainer Rollinger / Josef Schneeberger: Handbuch der Künstlichen Intelligenz,
Oldenbourg 2003 (4. Auflage), ISBN 3-486-27212-8

Stuart Russell / Peter Norvig: Artificial Intelligence: A Modern Approach,
Pearson 2003 (2. Auflage), ISBN 0-13-080302-2
Internet: http://aima.cs.berkeley.edu/ ,
Stand: 25.05.2005, Abruf: 25.05.2005

Kim Marriot / Peter J.Stuckey: Programming with Constraint: An Introduction,
MIT Press 1998, ISBN 0-262-13341-5

Vijay Saraswat / Pascal von Hentenryk (Hrsg.): Principles and Practice of Constraint Programming,
MIT Press 1995, ISBN 0-262-19361-2

Brian Mayoh, Enn Tyugu, Jaan Penjam (Hrsg.): Constraint Programming
NATO ASI Series Vol. 131, Springer 1994, ISBN 3-540-57859-5

o.V.: The ECL^{i}PS^{e} Constraint Logic Programming System,
Internet: http://www.icparc.ic.ac.uk/eclipse/ ,
Stand: 16.12.2004, Abruf: 08.05.2005

Roman Bartak: Online Guide to Constraint Programming,
Internet: http://kti.mff.cuni.cz/~bartak/constraints/ ,
Stand 14.03.2005, Abruf 20.05.2005