Projektbericht

Projekt „Constraint Programming“ bei Stefan Frank
Sommersemester 2006
Fachgebiet Übersetzerbau und Programmiersprachen
Technische Universität Berlin

Bernhard Beschow bbeschow@cs.tu-berlin.de
Reiner Czerwinski reinerc@cs.tu-berlin.de
Katrin Lang langk@cs.tu-berlin.de

Eliminate all other factors,
and the one which remains
must be the truth.
A. Conan Doyle

Zusammenfassung

TUCS ist ein in Common Lisp implementierter Constraint Solver. Er arbeitet auf prinzipiell beliebigen diskreten finiten Domains, wobei zumeist nur effiziente Algorithmen für Integer-Domains existieren.

Die Ziele, die wir mit dem Solver verfolgten waren:

- ein hybrides Design, das für unterschiedliche Problemstellungen unterschiedliche Lösungsstrategien zulässt
- Flexibilität, leichte Erweiterbarkeit
- komfortable Handhabung

Implementierungsschwerpunkte lagen bei:

- der Suchraumeinschränkung durch rückblickende (Look-back-) Strategien
- der Implementierung von Graphalgorithmen (All-Different/Global-Cardinality)

Inhaltsverzeichnis

1 Einleitung........ 3
2 Die Modellierung der Problemstellung........ 3
 2.1 Der Domain-Store........ 4
 2.2 Der Constraint-Store und der implizierte Constraint-Graph........ 4
 2.3 Methoden........ 5
 2.4 Indirekte Indizierung........ 5
3 Strategien........ 5
 3.1 Strategien zur Variablenauswahl........ 5
 3.2 Look-back-Strategien........ 6
  3.2.1 Backjumping........ 6
  3.2.2 Backmarking........ 8
  3.2.3 Implementierung der Look-back-Strategien........ 9
 3.3 Look-ahead-Strategien........ 9
 3.4 Konsistenz-Strategien........ 10
4 Die Klasse Constraint........ 11
5 Die wichtigsten Makros........ 13
 5.1 Parsen eines Lambda-Ausdrucks 13
 5.2 Code-Generierung 13
6 Binarisierung........ 13
7 Die Benutzung des Programms........ 14
8 Vorformulierte Probleme........ 15
 8.1 Färbbarkeit eines Graphen........ 15
 8.2 N-Damen........ 16
 8.3 Send-More-Money........ 16
 8.4 Sudoku........ 16
 8.5 Scheduling-Problem........ 17
9 Zeitmessungen........ 17
 9.1 N-Queens........ 17
  9.1.1 Nicht-Binarisiert........ 18
  9.1.2 Binarisiert........ 18
 9.2 Send-More-Money........ 18
 9.3 Sudoku........ 19
10 Erfahrungsbericht 20
11 Fazit........ 21
12 Quellen und Links........ 22

1 Einleitung

Wie effizient ein Constraint Satisfaction Problem (CSP) gelöst werden kann, hängt zum Einen davon ab, dass für die einzelnen Constraints Strategien zur wirksamen (teilweisen) Herstellung der lokalen Konsistenz gewählt werden können. So kann etwa eine Einschränkung der Grenzen der Domains insgesamt effizienter sein als die Herstellung vollständiger lokaler Konsistenz.

Zum anderen sollte auch das Aufbauen des Suchbaums parametrisierbar sein.

Ausgehend von diesen Forderungen formulierten wir ein hybrides CSP als:

1. ein *Domain-Store*, wo für jede Variable die einzuschränkende Domain verzeichnet ist
2. ein *Constraint-Store*, der einen Graphen mit den Variablen als Knoten impliziert
3. eine Reihe von *Strategien.*

Diese sind im Einzelnen:

- *Heuristiken* für die Auswahl der nächsten zu testenden Variablen
- *Look-back-Strategien*, die rückblickend die Suchbaumtraversierung gestalten
- *Look-ahead-Strategien*, die lokale Konsistenz in unterschiedlichem Grad propagieren
- *Consistency-Strategien*, die im Grad der hergestellten lokalen Konsistenz variieren

Zudem sollten diese Strategien zur Laufzeit austauschbar sein, um sich an eine möglichst ideale Problemformulierung herantasten zu können.

Mit unserer Modellierung haben wir beabsichtigt, ein flexibles Framework zu bieten, das leicht erweiterbar ist.

Kapitel 2 wird die Modellierung eines CSP erläutern, Kapitel 3 wird auf die einzelnen Lösungsstrategien näher eingehen, Kapitel 4 stellt die Klasse Constraint vor.

Die von uns verwendeten Makros werden in Kapitel 5 beschrieben, Binarisierung in Kapitel 6.

Kapitel 7 erklärt die Handhabung unseres Programms, Kapitel 8 schliesslich erläutert die von uns mitgelieferten Beispielmodellierungen, während Kapitel 9 deren Laufzeit bei unterschiedlicher Strategienwahl diskutiert.

In Kapitel 10 berichten wir über unsere Erfahrungen und die Auswirkungen, die die Wahl der Implementierungssprache Common Lisp auf unser Projekt hatte; in Kapitel 11 schließlich ziehen wir das Fazit unserer Arbeit.

2 Die Modellierung der Problemstellung

Dieses Kapitel erklärt die grundsätzliche Implementierung eines wie in Kapitel 1 formulierten CSP.

Die Klasse CSP wie in Abbildung 1 gezeigt hat im wesentlichen 6 Felder oder Slots:

- den initialen Domain Store
- den Constraint-Store mit impliziertem Graph
- eine Reihe von Strategien wie in Kapitel 1 beschrieben als Strategie-Objekte

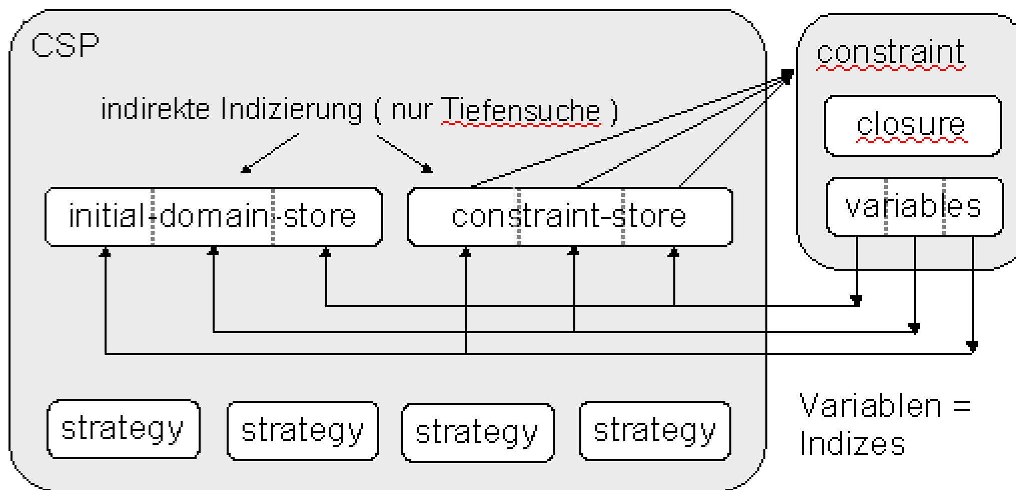

Abbildung 1: Die Modellierung eines CSP

Zusätzlich gibt es:

- einen Slot, der angibt, ob nur eine oder alle Lösungen gefunden werden sollen
- einen Slot, der angibt, ob die Lösung in menschlich besser erfassbarer Form auf den Bildschirm ausgedruckt werden soll
- einen Slot, der das Ergebnis des letzten Aufrufs enthält

2.1 Der Domain-Store

Der Domain-Store ist ein Vektor der Länge <Variablenanzahl>. Variablen sind Indizes in diesen Domain-Store. Zu jedem Variablenindex findet sich im Domain-Store die zugehörige einzuschränkende Domain als Liste. Diese Datenstruktur wird beim Traversieren des Suchbaums komplett als Kopie auf den Programmstack gelegt und bildet den aktuell gültigen Domain-Store.

Die Anzahl der Variablen bleibt während der Lebenszeit eines CSP unverändert. In weitergehenden Implementierungen könnte es aber nötig werden, den Domain Store zu erweitern, etwa wenn man allgemeine Constraints binarisieren möchte, oder zusammengesetzte Polynome für Constraints zulassen möchte, was beides das Zufügen von neuen Knoten erfordert.

2.2 Der Constraint-Store und der implizierte Constraint-Graph

Der Constraint-Store ist wie der Domain-Store ein Vektor der Länge <Variablenanzahl>. Für jeden Variablenindex enthält er ebenfalls Listen, welche jedoch Zeiger auf diejenigen Constraints enthalten, an denen die Variable beteiligt ist.

Ein Constraint in seiner einfachsten Form besteht aus einer Closure und einer Liste aller Variablen, die an ihm beteiligt sind. Optional ist eine Konsistenzstrategie.

Beliebige Lambda-Ausdrücke in Lisp-Syntax können als Closures für Constraints angegeben werden, sofern sie in der Anzahl der Parameter mit der Länge der Variablenliste übereinstimmen und einen bool'schen Wert zurückgeben. Es existiert dann auf jeden Fall eine anwendbare Konsistenz-Strategie, eventuell ist aber keine *effiziente* Konsistenz-Strategie vorhanden.

Die Variablenliste eines Constraints ist eine Liste von Ganzzahlen, die als Indizes in den aktuell auf dem Stack liegenden Domain-Store und in den Constraint-Store dienen. Wie bereits erwähnt muss die Variablenliste in Länge und Reihenfolge mit der Lambda-Liste der Closure übereinstimmen.

Somit ergibt sich ein impliziter Constraint-Graph.

Constraints können nachträglich zum Constraint-Store hinzugefügt werden, aber nicht ohne Wissen über die unterliegende Klassenstruktur und die konkreten Datenstrukturen abgeändert oder entfernt werden. Insbesondere kann der Constraint-Store initial leer sein.

2.3 Methoden

Die wichtigste Methode, die auf die Klasse CSP spezialisiert, ist die Methode `solve`, die alle benötigten Datenstrukturen initialisiert und daraufhin eine Tiefensuche aufruft. Diese bildet das eigentliche Rückgrat des Solvers. Aus diesem heraus werden wiederum generische Funktionen aufgerufen, deren Methoden auf die im CSP gespeicherten Strategien spezialisieren. Strategien können jederzeit ausgewechselt werden ermöglichen es so, das Verhalten des Solvers anzupassen.

Durch Subklassenbildung und Methodenkombination lassen sich Strategien komfortabel erweitern und/oder kombinieren.

2.4 Indirekte Indizierung

Sowohl der Domain-Store als auch der Constraint-Store werden in der Tiefensuche indirekt indiziert, um die Implementierung leistungsfähiger (dynamischer) Variablenauswahlheuristiken (siehe Kapitel 3.1) zu ermöglichen. Strategien müssen sich mit dieser indirekten Indizierung nicht befassen, mit Ausnahme der Look-Back-Strategien, falls für diese die Unterstützung (dynamischer) Variablenauswahlheuristiken möglich und erwünscht ist.

3 Strategien

Strategien sind als Klassen implementiert. Grund hierfür war einerseits die einfache Erweiterbarkeit/Kombinierbarkeit von Strategien, andererseits die Notwendigkeit einiger Strategien, eigene Datenstrukturen zu verwalten.

3.1 Strategien zur Variablenauswahl

Prinzipiell gibt es zwei Möglichkeiten, Variablen auszuwählen:

1. anhand der Anzahl von Constraints, durch die eine Variable beschränkt ist. Dies ist eine sog. Fail-first-Heuristik, d.h. es wird versucht, potentiell schneller scheiternde Möglichkeiten bereits weit oben im Suchbaum abzuschneiden.
2. nach der Domain-Grösse. Hier gilt: je kleiner die Domain, desto schneller wird eine potentielle Lösung gefunden. Es handelt sich also um eine Success-First-Heuristik.

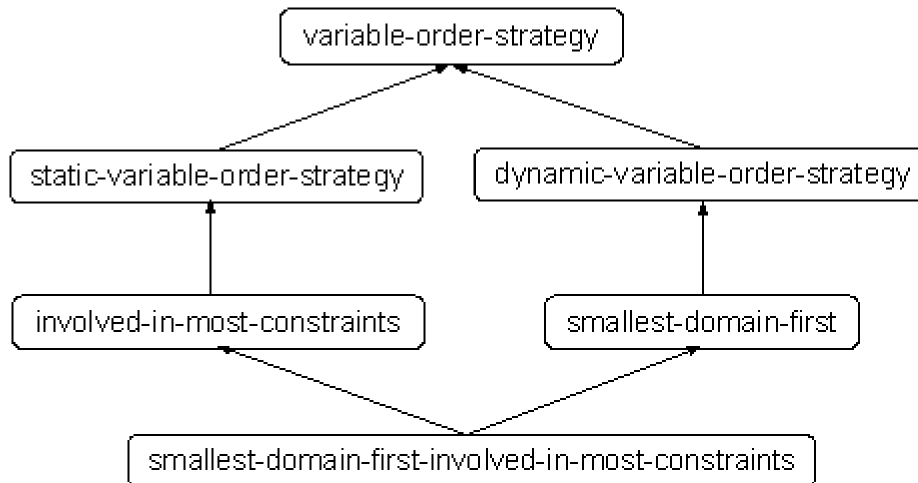

Abbildung 2: Klassenhierarchie der Variablenauswahlheuristiken

Wir haben 3 Heuristiken implementiert (siehe Abbildung 2):

Eine davon funktioniert rein nach dem ersten oben vorgestellten Prinzip. Die Variablen werden vor Aufruf der Tiefensuche *einmal* nach der Anzahl der Constraints, an denen die Variablen beteiligt sind, sortiert. Es handelt sich um eine *statische* Variablenauswahlheuristik.

Die anderen beiden Heuristiken sind dagegen von *dynamischer* Natur, d.h. nach jedem Domain-Einschränkungsschritt wird neu entschieden, welche Variable als nächstes instanziiert werden soll.

Die erste dynamische Strategie wählt Variablen nach dem zweiten oben dargestellten Prinzip, d.h. anhand der Domaingrösse aus. Die andere bildet eine kombinierte Strategie, die zunächst die Variablen nach dem zweiten Prinzip sortiert und evtl. in Frage kommende Variablen mit gleicher Domaingrösse anschliessend noch einmal nach der ersten Strategie.

3.2 Look-back-Strategien

Bei den sog. Look-back-Strategien handelt es sich um rückblickende Strategien, die zum Ziel haben, wiederholtes Testen in der Tiefensuche zu vermeiden, und/oder den sog. Thrashing-Effekt zu vermeiden, der im Folgenden erklärt wird.

3.2.1 Backjumping

Backjumping kann bei der Baumtraversierung Knoten und deren Unterbäume auslassen, ohne dass sich das Endergebnis ändert.

Reines *Backtracking* dagegen fährt nach dem Abarbeiten eines Knotens mit dem Instanziieren des nächsten Werts (falls vorhanden) des vorigen Knotens fort (Abbildung 4). Dabei werden u.U. vergebens Knoten getestet, die gar nicht am Constraint beteiligt sind, das ein Scheitern verursacht hat. Diesen Effekt nennt man *Thrashing*.

Backjumping springt dagegen direkt zum das Scheitern verursachenden Knoten zurück, d.h. im einfachsten Fall zum nächsttieferen Knoten, der am Constraint beteiligt ist. (Abbildung 3). Diesen Knoten nennt man den sicheren Rücksprung.

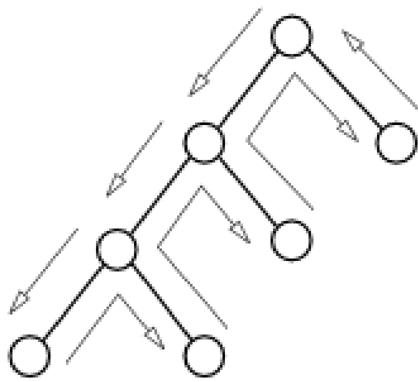

Abbildung 4: Backtracking

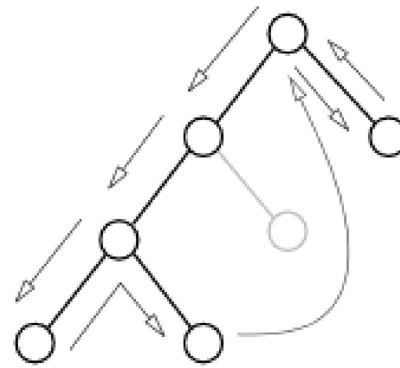

Abbildung 3: Backjumping

Sind mehrere Werte mit jeweils mehreren scheiternden Constraints vorhanden, so wird folgendermassen vorgegangen:

Abbildung 5 zeigt ein Beispiel für sieben Variablen, die bereits instantiiert wurden, wobei die gerade getestete Variable 7 zwei Werte annehmen kann.

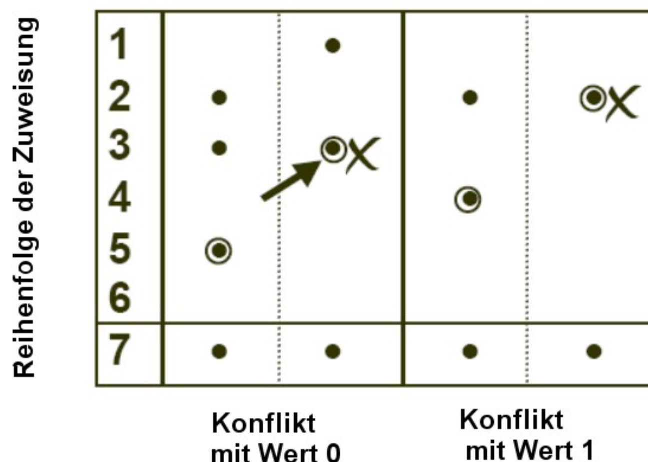

Abbildung 5: Sicherer Rücksprung

Grundsätzlich ist ein Rücksprung nur möglich, wenn *alle* Werte einer Variablen *vergebens* getestet wurden. Es wird dann der am wenigsten tief im Suchbaum befindliche sichere Rücksprung aller Werte gewählt.

Unterschiedliche Backjumping-Algorithmen unterscheiden sich in der Tiefe des gefundenen sicheren Rücksprungs. Der erhöhte Aufwand, mit dem dieser berechnet werden muss, wird durch die erzielte Suchraumeinschränkung oft aufgehoben.

Wir haben den von Gaschnig entwickelten, in Abbildung 4 illustrierten Algorithmus implementiert, der für jede Variable v_i, $i<=k$ nacheinander die Konsistenz der für jedes i anwendbaren Constraints für folgende Zuweisungen überprüft:

$v_1 = w_1$ $\quad v_{k+1} = w_{k+1}$

$v_1 = w_1 \quad v_2 = w_2$ $\quad v_{k+1} = w_{k+1}$

...

$v_1 = w_1 \quad v_2 = w_2 \quad \ldots \quad v_k = w_k \quad v_{k+1} = w_{k+1}$

Der Index $i <= k$, an dem eines der getesteten Constraints inkonsistent ist, bildet den sicheren Rücksprung.

Diese Überprüfung wird für jeden Wert w_j der Variable x_{k+1} durchgeführt. Wiederum ist der sichere Rücksprung der am wenigsten tief im Suchbaum befindliche Index aller sicheren Rücksprünge der Werte w_j.

3.2.2 Backmarking

Backmarking hat im Gegensatz zu Backjumping nicht das Ziel, den Suchraum einzuschränken, sondern wiederholtes Testen zu vermeiden. Abbildung 6 zeigt eine Situation, wo im Suchbaum zum wiederholten Mal eine Instantiierung von $\mathbf{v_k}$ mit dem Wert **b** erreicht wird. Teile des Suchbaums, nämlich bis zum Index **i** bleiben aber gleich. Merkt man sich nun in einer Datenstruktur für jede Variable $\mathbf{v_i}$ und jeden derer Werte $\mathbf{w_{ij}}$ den sicheren Rücksprung **r** und zudem den Index **c** im Suchbaum, wo zuletzt eine Änderung, d.h. Instanziierung vorgenommen wurde, so kann auf erneutes Testen (teilweise) verzichtet werden.

Im Falle **r > c** sind alle Constraints zwischen den Variablen $\mathbf{v_1, ..., v_c, v_k}$ weiterhin konsistent, eine erneute Überprüfung dieser Baumschichten ist nicht nötig. Andernfalls ist die Situation weiterhin inkonsistent und es muss gar nicht geprüft werden.

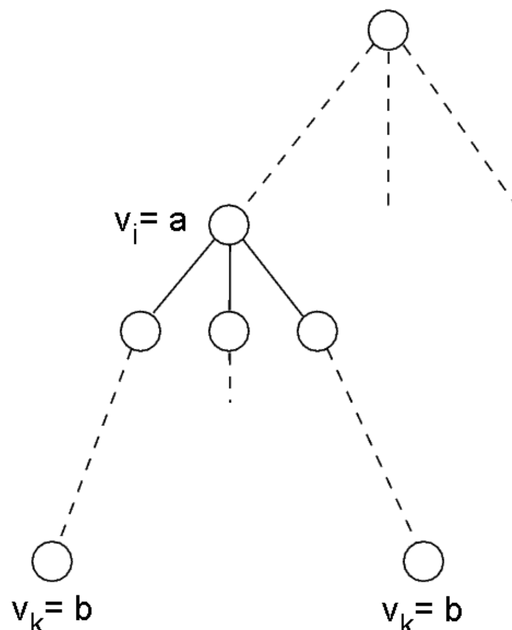

Abbildung 6: Backmarking

3.2.3 Implementierung der Look-back-Strategien

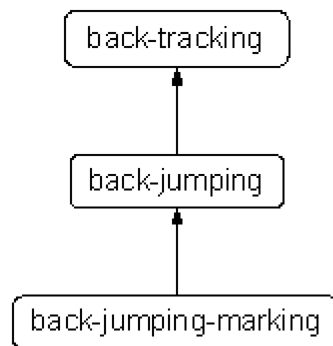

Abbildung 7: Klassenhierarchie der Look-back-Strategien

Da beide vorgestellten Verfahren auf einem sicheren Rücksprung basieren, wurde Backmarking nicht separat implementiert, sondern nur als hybride Backjumping-marking-Strategie, die Backjumping erweitert. Backjumping funktioniert im Gegensatz zu Backjumping-marking nicht nur auf Ganzzahl-Domains.

Von der Tiefensuche aufgerufen wird die Methode `find-conflict`. Sie wählt die anwendbaren Constraints für Variablenindex $\mathbf{v_{k+1}}$ aus. Anwendbar sind Constraints, wenn deren Variablen vollständig instanziiert sind. Beim Testen auf Konsistenz werden die in den anwendbaren Constraints gespeicherten Closures aufgerufen.
Scheitert der Konsistenztest, so wird die Methode `backjump` aufgerufen, die ein Integer `i` zurückgibt, also einen Variablenindex v_i im Suchbaum, zu dem zurückgesprungen wird. Bei einem Erfolg wird `k+1` zurückgegeben.

Die `backjump` -Methode, die auf die Klasse `backtracking` spezialisiert, gibt einfach `k` zurück.

Die auf `backjumping` bzw `backjumping-marking` spezialisierenden Methoden berechnen den Rücksprung auf die in den beiden vorangehenden Unterkapiteln vorgestellte Weise.

Die `find-conflict`-Methode spezialisiert im übrigen zusätzlich zu einer Look-Back-Strategie auch auf eine statische Variablenauswahlstrategie und auf eine dynamische Variablenauswahlstrategie, um effizientere Methoden ohne indirekte Indizierung bereitstellen zu können, bzw zu verhindern, dass Backmarking mit einer dynamischer Variablenauswahlstrategie aufgerufen wird.

3.3 Look-ahead-Strategien

Wir haben zwei Look-ahead-Strategien implementiert:

1. Forward Checking
2. AC3

Forward Checking unterscheidet sich von der AC3-Strategie in der Hinsicht, dass nur lokale Konsistenz für die direkt an einer Variablen beteiligten Constraints hergestellt werden, während AC3 die lokale Konsistenz durch den gesamten Constraint-Graphen propagiert. Zur Verwaltung der als nächstes zu betrachteten Constraints benutzt AC3 eine von uns implementierte Queue, die ein Attribut, also ein Slot der Klasse AC3 ist.

Auf die Look-ahead-Strategien spezialisiert die Methode `confine-domains`, die für jedes betrachtete Constraint die Methode `local-consistency` (siehe nächstes Kapitel) aufruft.

3.4 Konsistenz-Strategien

Abbildung 8 zeigt die von uns implementierten Konsistenzstrategien.

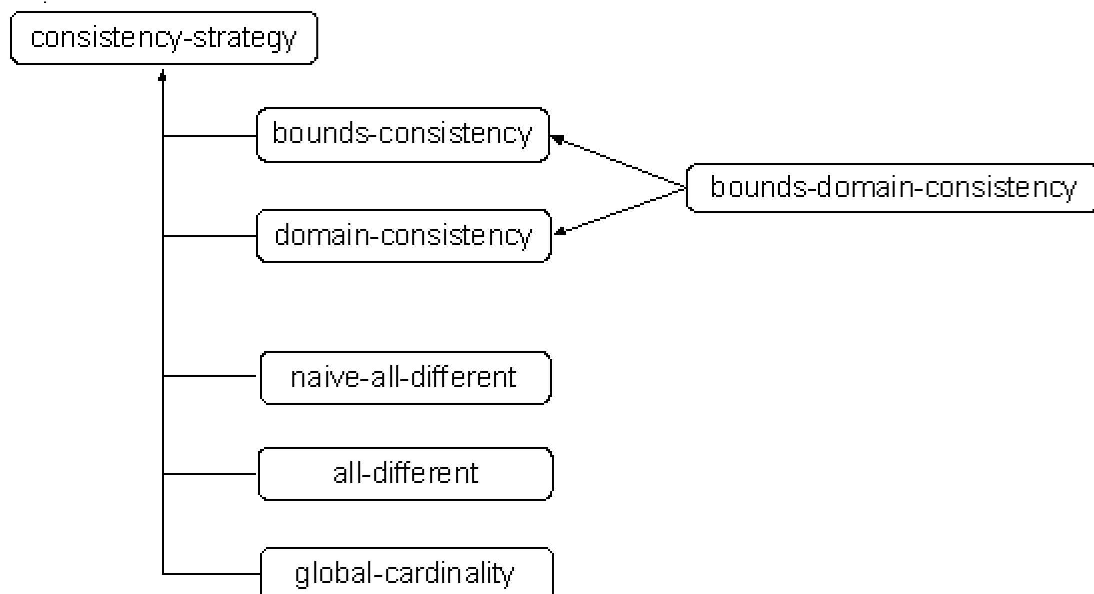

Abbildung 8: Klassenhierarchie der Konsistenz-Strategien

Bounds-Consistency schränkt nur die Grenzen der Domains der am Constraint beteiligten Variablen ein, während Domain-Consistency vollständige lokale Konsistenz herstellt, zum Preis eines im allgemeinen Falle exponentiellen Aufwandes. Sie lohnt sich daher bei n-stelligen Constraints nur für kleine n.

Domain-Consistency baut für jede Variable und für jeden Wert der Domain dieser Variable den Lösungsbaum auf und testet mit der Closure des Constraints. Sobald die Überprüfung einmal konsistent war, wird die Suche abgebrochen und der Wert verbleibt in der Domain. Schlägt die Suche fehl, so wird der Wert entfernt.

Diese Implementierung hat zum Vorteil, dass für jedes Constraints zumindest *eine* Konsistenzstrategie vorhanden ist, die zudem als einzige Konsistenzstrategie nicht nur Integer-Domains voraussetzt. Hat man beispielsweise ein CSP mit n effizient lösbaren Constraints und nur einem Constraint, das nicht mit anderen Strategien effizient lösbar ist, so lässt sich das Gesamt-CSP im Rahmen der Möglichkeiten immer noch effizient lösen, wenn man für dieses eine Constraint Domain-Consistency verwendet. Alternativ hätte man auch eine Dummy-Konsistenzstrategie einführen können, so dass für das besagte Constraint effektiv nur Backtracking benutzt wird.

Es wurde außerdem eine hybride Strategie implementiert, die zuerst die Grenzen einschränkt, und auf die eingeschränkten Domains eine Domain-Konsistenz aufruft. Diese Strategie konnte durch Methodenkombination auf denkbar einfache Weise implementiert werden.

Weiterhin implementiert wurden ein naives All-Different, ein Graph-All-Different nach Hopcroft-Carp und eine Global-Cardinality-Strategie.

Die naive All-Different-Strategie sucht einelementige Domains und entfernt die darin enhaltenen Werte aus den übrigen Domains solange, bis ein Fixpunkt erreicht ist.

Bei der graphbasierten All-Different-Strategie werden aus dem vom Constraint induzierten Graphen alle Kanten mit der Funktion `remove-not-matched-arcs` entfernt, die nicht zu einem maximalen Matching gehören. Der Algorithmus wurde funktional implementiert. Genauere Details finden sich in den Folien zur Veranstaltung.

Die Global-Cardinality-Strategie funktioniert analog zum graphbasierten All-Different. Es werden zwei Graphen erzeugt, in denen die Anzahl der Werte in den Domains jeweils mit der oberen bzw. unteren Schranke der Domains multipliziert wird, sodass die Konsistenz des Constraints anhand der Existenz der maximalen Matchings in den Graphen entschieden werden kann.

4 Die Klasse Constraint

Die Superklasse aller Constraints ist die Klasse Constraint. Sie besitzt 4 Slots:

1. eine Closure
2. eine Variablenliste
3. eine Konsistenzstrategie, die nil sein kann
4. eine Default-Konsistenzstrategie, die zur Anwendung kommt, falls keine explizite Konsistenzstrategie angegeben wurde. Diese ist als `:allocation :class` (d.h. ein Speicherplatz für alle Instanzen) Slot deklariert und als einziges Attribut eines Constraints jederzeit ohne weiteres veränderbar.

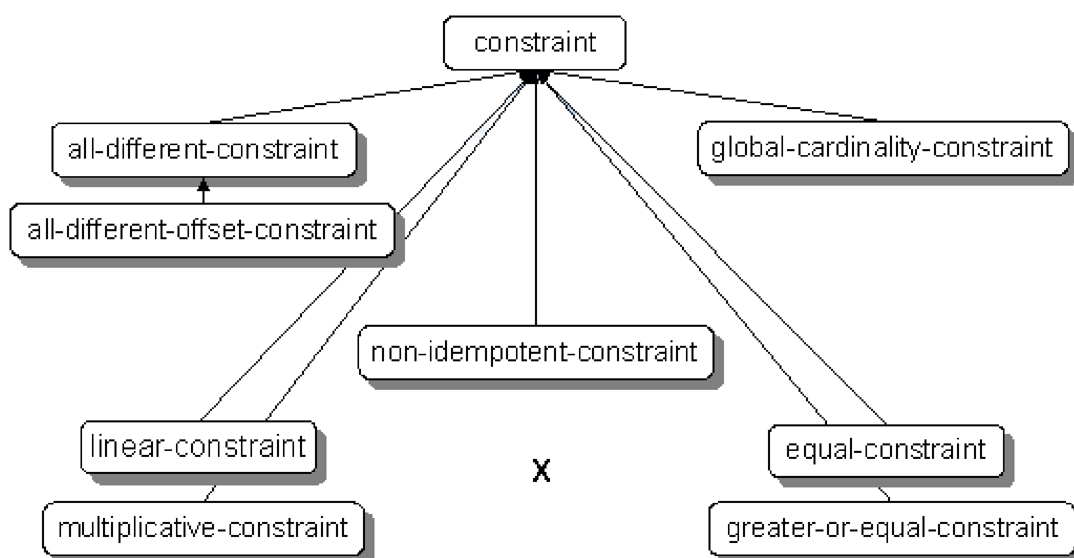

Abbildung 9: Klassenhierarchie der Constraints

Abgeleitet von dieser Klasse sind `all-different-constraint`, `global-cardinality-constraint`, `non-idempotent-constraint` und die Komplemente `linear-constraint` und `multiplicative-constraint` bzw `eql-constraint` und `greater-eql-constraint`.

Diese beiden Paare sind nicht zur direkten Instanziierung vorgesehen, sondern nur deren Subklassen (und optional die der Klasse `non-idempotent-constraint`) `linear-eql-constraint`, `multiplicative-eql-constraint`, `linear-greater-eql-constraint`, `multiplicative-greater-eql-constraint`, `non-idempotent-linear-eql-constraint`, `non-idempotent-multiplicative-eql-constraint`, `non-idempotent-linear-greater-eql-constraint` und `non-idempotent-multiplicative-greater-eql-constraint`. Aus Gründen der Übersichtlichkeit wurde dies in der Abbildung durch den x-Operator dargestellt.

Die Komplexität der Klassenstruktur hätte sich verringern lassen, hätte man jedes `eql-constraint` durch zwei `greater-eql-constraints` ersetzt. Aus Laufzeiteffizienzgründen haben wir uns jedoch dagegen entschieden, da dies einen verdoppelten Aufwand beim Abprüfen bzw Herstellen der Konsistenz der Constraints bedeutet hätte. Ohnehin übernimmt die korrekte Instanziierung der richtigen Klasse ein Makro (siehe folgendes Kapitel).

Lineare Constraints besitzen zwei zusätzliche Slots, in denen die Koeffizienten als Liste von Integern und ein `k`, berechnet aus den Konstanten c_i einzelnen ebenfalls als Integer gespeichert sind. Ein Ausdruck der Form (Vertauschungen von Faktor a_i und Variable x_i sowie von Konstanten c_1 und Ausdrücken $(* \ a_i \ x_i)$ sind erlaubt):

$$(eql \ (+ \ (* \ a_1 \ x_1) \ldots \ (* \ a_k \ x_k) \ c_1 \ \ldots \ c_l) \ (+ \ (* \ a_{k+1} \ x_{k+1}) \ldots (* \ a_m \ x_m) \ c_{l+1} \ \ldots \ c_n))$$

wird dazu umgeformt in:

$$\sum koeff_i * x_i = k \quad \text{, womit gilt} \quad x_j = (k - \sum_{i \neq j} koeff_i * x_i) / koeff_j$$

Die multiplikativen Constraints erlauben folgende Syntax für die Closure.

```
(lambda (x y z) (eql (* (* a x) (* b y)) (* x z))
```

Diese Syntax wurde als für unsere Zwecke ausreichend angesehen, da sämtliche Polynome durch Einführen von Hilfsvariablen in solche multiplikativen Ausdrücke umfgeformt werden können und das Bereitstellen beliebiger Polynome auch mit Grenzenkonsistenz nicht effizient wäre.

Die Methode `establish-consistency` ist die eigentliche Methode, die lokale Konsistenz herstellt und spezialisiert auf 2 Klassen: auf ein `constraint` und eine `consistency-strategy`. Die Methode `local-consistency` ruft diese Methode lediglich geeignet parametrisiert nach den im Constraint gespeicherten Konsistenz-Strategien auf. Eine **:around-**Methode zu `local-consistency` spezialisiert auf `non-idempotent-constraint`**.** Sie überprüft auf einen Fixpunkt und ruft, solange dieser noch nicht erreicht ist, die Hauptmethode auf.

Für All-Different-Constraints und All-Different-Offset-Constraints sind als Konsistenzstrategie (außer `domain-consistency`) `naive-all-different` und `all-different` wählbar,

für Global-Cardinality-Constraints domain-consistency und global-cardinality. All-Different-Offset wurde durch eine :around-Methode zu establish-consistency implementiert.

Für lineare und multiplikative Constraints kann unter bounds-consistency und domain-consistency bzw. bounds-domain-consistency gewählt werden.

5 Die wichtigsten Makros

Makros wurden von uns verwendet, um Constraints automatisch zu instanziieren bzw Code für die Closures spezieller zu instantiierter Constraints zu generieren und damit die Benutzung unseres Programms zu vereinfachen.

5.1 Parsen eines Lambda-Ausdrucks

Das Makro make-constraint instantiiert ein Constraint oder eine seiner Subklassen, indem es einen übergebenen Lambda-Ausdruck parst. Es müssen weiterhin eine Variablenliste und der Name einer Konsistenzstrategie als Symbol (oder nil für Verwenden der Default-Konsistenzstrategie) angegeben werden, und es muss spezifiziert werden ob das Constraint idempotent ist (T) oder nicht (nil).

Der Parser stellt Ausdrücke der Form

```
(lambda (x y z) (eql (* a x) (* (* b y) (* c z))))
```

in für multiplikative Constraints legale Ausdrücke

```
(lambda (y z x) (eql (* (* -1 (* b y) (* c z))) (* -1 (* a x))))
```

um und vertauscht auch die Variablenliste entsprechend.

Für lineare Constraints werden die Koeffizienten und k berechnet. Dabei werden für <- bzw **<=**-Ausdrücke für die ein greater-eql-constraint instantiiert werden soll, die Koeffizienten und k negiert. Für **<**- und **>**-Ausdrücke wird anschließend nochmals 1 abgezogen.

Genügt der Lambda-Ausdruck nicht den Anforderungen für ein lineares oder multiplikatives Constraint (dazu genügen bereits auch Faktoren, die keine Ganzzahlen sind), so wird eine einfache Instanz der Klasse constraint mit domain-consistency als Konsistenzstrategie erzeugt.

5.2 Code-Generierung

Die Makros make-all-different-constraint und make-global-cardinality-constraint instanziieren und initialisieren das jeweilige Constraint. Für make-all-different-constraint ist dazu lediglich die Angabe der Variablenliste (und ggf. einer Konsistenzstrategie oder nil) vonnöten. Bei Global-Cardinality-Constraints ist zu beachten, dass die Domains aller beteiligten Variablen gleich sein müssen; make-global-cardinality-constraint akzeptiert neben den beteiligten Variablen eine minimale Kardinalität und eine

maximale Kardinalität jeweils als Integer, die allen Variablen gemeinsame initiale Domain als Liste und den Namen einer Konsistenzstrategie als Symbol oder `nil`.

6 Binarisierung

Binarisierung wurde von uns für *globale Constraints* implementiert, d.h. Constraints, die zwischen je zwei Variablen existieren. Zu diesem Zweck werden $\binom{Variablenanzahl}{2}$ neue Constraints genieriert. Dies bringt, wie man sich leicht vorstellen kann, enorme Geschwindigkeitsvorteile, wenn keine effiziente Consistency-Strategie vorhanden ist, da in der Tiefensuche Äste bereits früh im Suchbaum abgeschnitten werden können.

Das Binarisieren geschieht beim Zufügen eines Constraints zum CSP. Um ein Constraint zu binarisieren reicht es aus, eine leere Variablenliste und eine 4-stellige Funktion anzugeben, die als Parameter je zwei Variablenindizes und zwei Werte für diese Variablen akzeptiert. Erstere werden beim Hinzufügen des Constraints zum CSP gecurried, so dass sich eine zweistellige Closure ergibt.

7 Die Benutzung des Programms

Alle Dateien des Programms befinden sich im Ordner `tucs`. Zum Kompilieren des Programms wird `load-tucs` aufgerufen. Dazu muss die Datei „`tucs/load-tucs.lisp`“ geladen werden.

Der erste Schritt bei der Modellierung eines CSP ist es, eine Instanz der Klasse CSP zu erzeugen. Der **`:domains`**-Slot kann als Liste der Form (*<Variablenanzahl> <Werteanzahl>*) angegeben werden. Eine **`:after`**-Methode zu `initalize-instance` initialisiert einen Domain-Store der Größe <*Variablenanzahl*>. durch identische Domains mit den Werten von *0* bis <*Werteanzahl-1*>. Es ist jedoch auch die direkte Angabe eines initialen Domain-Stores als Liste von Listen, also Domains, möglich. Die Constraints können als Liste mit dem Schlüsselwort **`:constraints`** angegeben werden oder aber sie können nachträglich mit Hilfe der Methode `add-constraint`, die auf ein CSP und ein `constraint` spezialisiert, zu einem `CSP` hinzugefügt werden.

Die Methode `list-all-constraints` listet alle im CSP gespeicherten Constraints auf, deren Slots dann gezielt ausgelesen und gesetzt werden können. Von dieser Low-Level-Bearbeitungsmöglichkeit solte man aus Sicherheitsgründen aber absehen und stattdessen lieber ein neues CSP erzeugen und `make-constraint` benutzen, um dafür neue Constraints zu generieren.

Ausserdem können bei der Instanziierung eines CSP folgende Initialisierungsargumente angegeben werden (Das Symbol nach jedem Schlüsselwort bezeichnet den Initialisierungswert, falls das Initialisierungsargument nicht angegeben wurde):

```
:static-variable-order-strategy 'no-static-sort
:dynamic-variable-order-strategy 'no-dynamic-sort
:look-back-strategy 'BT
:look-ahead-strategy 'no-look-ahead
:default-consistency-strategy 'domain-consistency
:all-solutions T
:print-result nil
```

Die folgenden Methoden erlauben es, eine Strategie nachträglich zu ändern. Sie spezialisieren sämtlich auf die Klasse `CSP` und erhalten ein eine Strategie bezeichnendes Symbol als Parameter:

- `set-svos` setzt die statische Variablenauswahlstrategie
- `set-dvos` setzt die dynamische Variablenauswahlstrategie, falls keine Backmarking-Strategie als Look-back-Strategie gesetzt ist
- `set-lbs` setzt die Look-Back-Strategie
- `set-las` setzt die Look-Ahead-Strategie
- Die Default-Konsistenzstrategie kann jederzeit durch die Methode `set-dcs` geändert werden. Dazu besitzt die Klasse CSP einen entsprechenden Slot, der zu jedem Zeitpunkt den gleichen Wert hat wie der entsprechende klassenalloziierte Slot der Constraints. Falls der Constraint Store initial leer ist, wird beim Hinzufügen eines Constraints dessen `default-consistency-strategy`-Slot aus dem gleichlautenden slot des CSP initialisiert.

Die Methoden `set-all-solutions` und `set-print-solutions` akzeptieren ein CSP und einen bool'schen Wert als Parameter und bestimmen, ob alle Lösungen gefunden werden sollen bzw ob diese durch eine spezielle print-Methode `print-solutions` ausgedruckt werden soll. Ein direkter Aufruf dieser Methode gibt das letzte Resultat des übergebenen CSP auf den Bildschirm aus.

Die Methode `solve` schließlich löst das in der vorgestellten Weise formulierte CSP und gibt die Ergebnisse als Liste von Listen von Werten in der Reihenfolge der angegebenen Variablen zurück.

8 Vorformulierte Probleme

Wir haben eine Reihe von Problemen als spezielle Subklassen von `CSP` vorformuliert, was es uns ermöglichte, für jede dieser Subklassen eigene Print-Methoden bereitzustellen.

Ein Makro parst eine aufs Problem abgestimmte Eingabe und initialisiert eine spezielle `CSP`-Instanz, wobei für die im CSP gespeicherten Constraints gegebenenfalls Code generiert wird.

8.1 Färbbarkeit eines Graphen

Dieses CSP repräsentiert das Problem, einen Graphen mit n Farben einzufärben.

Ein Beispiel:

Die Landkarte Australiens soll mit 3 Farben Rot Grün und Blau gefärbt werden, so dass angrenzende Regionen unterschiedliche Farben besitzen.

Variablen entsprechen dabei Regionen mit den jeweiligen **Domains** (RED GREEN BLUE), und es existiert eine **Relation** „ungleiche Farbe“ zwischen je 2 angrenzenden Regionen (sieheAbbildungen 10 und 11)

Es wurden eine Problemstellung formuliert, die beliebige Graphen, angegeben als Liste von Zwei-Tupeln von Integern, die die jeweiligen Knoten bezeichnen, zwischen denen eine „ungleiche-Farbe“-Relation bestehen soll, und eine Liste von Farben als Symbole, nach diesen Farben einfärbt.

Beispielaufruf:

```
(make-color-graph-csp 3 '((0 1) (1 2) (2 0)) '(red green blue))
```

Zur Demonstration wurde das oben beschriebene Problem implementiert, die Landkarte Australiens mit drei Farben zu färben. Es steht eine eigene Print-Methode bereit. Das Problem ist allerdings so einfach, dass es sich mit reinem Backtracking so schnell lösen lässt, dass eine Zeitmessung nicht sinnvoll möglich ist.

Abbildung 10: Landkarte Australiens

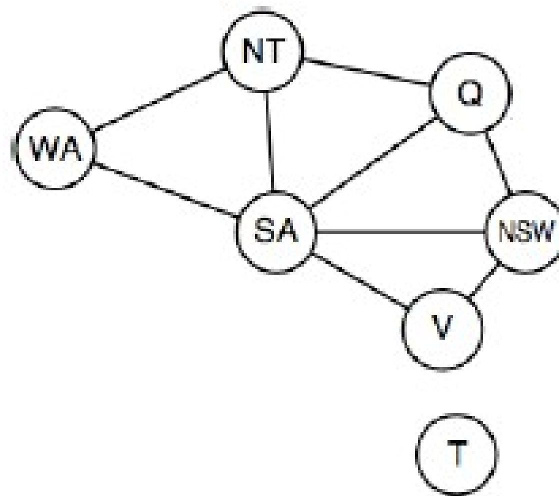

Abbildung 11: Australien als Graph

8.2 N-Damen

Das N-Damen-Problem repräsentiert das Problem, n Damen so auf einem Schachbrett der Größe **n x n** zu platzieren, dass sie sich nicht bedrohen, d.h. sich nicht auf derselben Zeile, Spalte, oder Diagonalen befinden.

Es wurde eine binarisierte und als nicht-binarisierte Version implementiert, da uns anfangs noch kein All-Different zur Verfügung stand. Beide Makros, `make-n-queens-csp` und `make-binarized-n-queens-csp` benötigen ein Integer `n` als Parameter, das die Dimension des Schachbretts und damit die Anzahl der zu platzierenden Königinnen angibt. Die Zeilen werden durch die unterschiedlichen Variablenindizes repräsentiert, die Lösungen sind die Spalten, auf denen die jeweiligen Damen platziert werden sollen.

Die nicht-binarisierte Version besitzt ein All-Different-Constraint für die Spalten und zwei All-Different-Offset-Constraints für die Diagonalen. Die Default-Konsistenzstrategie ist frei wählbar, Standard-Wert ist Naive-all-Different. Die Look-Ahead-Strategie ist initial AC3.

Die binarisierte Version dagegen benutzt nur reines Backtracking.

8.3 Send-More-Money

Send-More-Money bezeichnet das folgende Zahlen-Rätsel für die Zahlen von 0 bis 9:

$$
\begin{array}{r}
1000\,S + 100\,E + 10\,N + D \\
+\ 1000\,M + 100\,O + 10\,R + E \\
\hline
= 10000\,M + 1000\,O + 100\,N + 10\,E + Y
\end{array}
$$

Das Send-More-Money-CSP beinhaltet zwei unäre Constraints, die für `S` bzw `M` den Wert 0 ausschließen, ein All-Different-Constraint, das die Default-Konsistenzstrategie Naive-All-Different benutzt, und ein lineares Constraint, für das Bounds-Consistency vorgegeben ist.

8.4 Sudoku

Das Makro `make-sudoku-csp` akzeptiert ein als Liste übergebenes Sudoku der Form:

```
'(5 3 _ _ 7 _ _ _ _
  6 _ _ 1 9 5 _ _ _
  _ 9 8 _ _ _ _ 6 _
  8 _ _ _ 6 _ _ _ 3
  4 _ _ 8 _ 3 _ _ 1
  7 _ _ _ 2 _ _ _ 6
  _ 6 _ _ _ _ 2 8 _
  _ _ _ 4 1 9 _ _ 5
  _ _ _ _ 8 _ _ 7 9)
```

Das Ergebnis wird in derselben Form ausgedruckt.

Ein Sudoku-CSP besteht rein aus unären und All-Different-Constraints. Es kann als Default-Konsistenzstrategie zwischen Naive-All-Different und Graph-All-Different gewählt werden, wobei standardmäßig Graph-All-Different gesetzt ist.

Es wurde eine Funktion `read-sudoku` implementiert, die ein Sudoku aus einer txt-Datei einliest.

8.5 Scheduling-Problem

Scheduling bedeutet zeitliche Zuordnung von Jobs oder Prozessen, die beschränkte Ressourcen benötigen, unterschiedliche Längen, Startzeiten und Deadlines haben können. Für die Modellierung der vorhandenen Ressourcen wurden Global-Cardinality-Constraints benutzt.

Die EingabeParameter für `make-scheduling-csp` , das ein Scheduling-Problem erzeugt, sind:

`jobs`	Liste, die fuer jeden Job eine Liste der benötigten Betriebsmittel angibt
`resources`	Anzahl der Resourcen fuer jedes Betriebsmittel
`total-time`	Anzahl der gesamten Zeitschritte
`duration`	Benötigte Zeit (Zeitschritte) fuer jeden Job Default: 1 fuer alle Jobs
`begin`	Bereitstellungszeit für jeden Job Default: 0 für alle Jobs
`before`	Liste mit zeitlichen Abhängikeiten , wobei gilt : Job i muss vor Job j ausgeführt werden gdw. '(i j) in before enthalten Default : keine Abhängigkeiten
`dead-lines`	Liste von Deadlines fuer jeden job Default: Total-Time

Um die unterschiedliche Dauer von Jobs zu implementieren, wurden die Jobs in Teiljobs mit einer Zeiteinheit zerlegt und Constraints der Form: Startzeit von Job_{i+1} = Startzeit Job_i + 1 erzeugt.

Um im Ergebnis nur die ursprünglichen Jobs auszugeben, wird die initiale Jobzahl in einem Slot gespeichert.

9 Zeitmessungen

Es wurde die reine Laufzeit von 100 Aufrufen von `solve` gemessen und gemittelt. Die Zeitmessungen erfolgten ohne die Print-Ausgabe und jeweils für das Suchen aller Lösungen und werden in Sekunden angegeben.

Getestet wurde auf einem AMD Sempron 2,8 Ghz Notebook mit dem Compiler Clisp. Tests mit SBCL auf dem Server Pepita ergaben ähnliche Werte.

Alle Tests können durch Aufruf von `test-tucs` *<Durchläufe>* nachvollzogen werden.

9.1 N-Queens

Es wurde die Zeit für **n = 8** gemessen. Jede Variable ist an gleich vielen Constraints beteiligt, daher scheiden entsprechende Variablenauswahlstrategien aus. Auch Variablenauswahlstrategien, die nach Domaingröße auswählen, bringen bei dieser Problemstellung lediglich Overhead mit sich und wurden nicht getestet.

9.1.1 Nicht-Binarisiert

Look-back-Strategien wurden aufgrund der Tatsache, dass alle Constraints zwischen allen Variablen existieren, nicht getestet. Ein Rücksprung wäre damit nie tiefer als *<Variablenanzahl-2>*.

Getestet wurden:

— mit AC3

— mit Naive-All-Different **0,822s**

— mit All-Different **2,389s**

— mit Forward-Checking

— mit Naive-All-Different **0,965s**

— mit All-Different **1,546s**

Dies zeigt, dass die weniger mächtigen Strategien wie Forward Checking und Naive-all-Different bei einfachen Problemen von Vorteil sein können.

9.1.2 Binarisiert

Die binarisierte N-Damen-Version erreicht mit reinem Backtracking eine Zeit von **2,701s** und demonstriert, dass mit Binarisierung ein immerhin noch akzeptables Ergebnis erzielt werden kann, falls keine effiziente Konsistenzstrategie vorhanden ist.

Backjumping bzw Backmarking bringen bei N-Queens allerdings nur zusätzlichen Overhead mit sich. Die Zeiten sind **4.469s** bzw **4,485s**.

Look-Ahead-Strategien ergeben bei den binarisiert formulierten Constraints keinen Vorteil; mit AC3 und Domain-Consistency für beide Constraints ergaben sich **7,190s**, mit Forward Checking **5,339s**.

9.2 Send-More-Money

Unter Verwendung des AC3 ergaben sich folgende Zeiten:

Mit All-Different für das All-Different-Constraint und Bounds-Consistency für das lineare Constraint wurde das Problem in **0,028s** gelöst, mit Bounds-Domain-Consistency in **0, 059s.**

Mit Naive-All-Different dagegen ergaben sich **0,015s** bzw **0, 045s**.

Unter Verwendung von Forward Checking verschlechterten sich die Zeiten:

Mit All-Different für das All-Different-Constraint und Bounds-Consistency für das lineare Constraint wurde das Problem in **0,712s** gelöst, mit Bounds-Domain-Consistency in **0,742s**.

Mit Naive-All-Different und Bound-Consistency wurde eine Zeit von **0,559s**, mit Bounds-Domain-Consistency eine ähnliche Zeit von **0,585s** erzielt.

Domain-Konsistenz wurde nicht separat getestet, da die Lösung über 10 Minuten benötigte.

Look-Back- und Variablenauswahl-Strategien wurden nicht getestet, da Send-More-Money mit nur wenigen Backtracking-Schritten gelöst werden kann.

9.3 Sudoku

Es wurden ein einem Sudoku-Rätselbuch ein als einfach deklariertes Sudoku, ein mittelschweres, und ein schwieriges Sudoku entnommen.

Es wurde nur unter Verwendung des AC3 getestet. Da die Anzahl der Constraints für jede Variable gleich ist, kam eine Variablenauswahl nach diesem Kriterium nicht in Frage.

Look-back-Strategien wurden ebenfalls nicht getestet, da Backtracking bei All-Different nur in sehr begrenzten Ausmaß bei schwierigen Sudokus zum Zuge kommt.

Das einfache Sudoku

_	1	5	6	9	2	7	_	3
6	8	3	4	_	_	9	_	2
_	_	_	_	8	3	5	_	4
_	4	6	_	_	9	_	3	_
3	_	9	7	1	_	8	5	6
7	_	_	_	6	_	_	4	9
_	6	2	_	_	_	3	7	_
9	_	_	8	_	_	_	2	1
_	3	8	_	7	_	4	9	5

erzielte folgende Werte:

- mit naivem All-Different: **0.042 s**
- mit naivem All-Different und dynamischer Variablenauswahlstrategie: **0.091 s**
- mit Graph All-Different: **0.083 s**
- mit Graph-All-Different und dynamischer Variablenauswahlstrategie: **0.132 s**

Das mittelschwere Sudoku

_	_	_	1	_	_	_	_	_
_	_	8	_	_	_	_	1	_
6	5	_	3	8	_	4	9	_
_	_	_	_	_	_	9	_	3
_	_	_	8	_	3	_	4	_
3	7	2	4	_	6	5	8	_
_	_	_	_	_	_	_	3	9
_	3	7	_	_	_	6	_	_
_	2	_	6	_	9	1	5	_

erzielte folgende Werte:

- mit naivem All-Different: **0.056s**
- mit naivem All-Different und dynamischer Variablenauswahlstrategie: **0.105s**
- mit Graph All-Different: **0.155s**
- mit Graph-All-Different und dynamischer Variablenauswahlstrategie: **0.203s**

Das schwierige Sudoku

9	_	_	5	_	_	_	_	_
_	8	5	_	_	4	_	_	_
3	_	_	_	_	7	_	_	_
6	_	3	4	_	5	_	_	_
_	5	4	_	1	_	_	8	_
_	_	_	2	_	_	_	3	_
_	7	_	_	_	2	_	_	5
_	_	_	_	_	_	2	9	3
5	_	_	_	_	_	8	4	_

erzielte folgende Werte:

- mit naivem All-Different: **1.964s**
- mit naivem All-Different und dynamischer Variablenauswahlstrategie: **0.314s**
- mit Graph All-Different: **0.255s**
- mit Graph-All-Different und dynamischer Variablenauswahlstrategie: **0.304s**

10 Erfahrungsbericht

Die Wahl der Programmiersprache Common Lisp hatte in verschiedener Hinsicht Einfluss auf unsere Implementierung.

Common-Lisp ist ein objektorientierter Lisp-Dialekt und vereinigt als solcher funktionale Programmierung mit imperativen Aspekten wie Seiteneffekten. Dadurch konnten wir uns bei der Implementierung von Algorithmen aus der Welt der verschiedenen Programmierparadigmen geeignet „bedienen". Zum großen Teil ist der Solver funktional implementiert. Beispielsweise bei der Instantiierung von Variablen mit festen Werten in der Tiefensuche, sowie bei der Einschränkung der Domains mittels Look-ahead-Strategien wird hingegen auf Seiteneffekte zurückgegriffen und der aktuelle Domain-Store wird destruktiv verändert.

Beim Entwickeln und Testen kam uns der sog. *Read-eval-print-Loop (REPL)* Lisps zugute. Er erlaubt ein schnelles Weiterentwickeln von Teilen der Funktionalität ohne langwierige Änderungs-Kompilierungs-Test-Zyklen. Aus dem Read-eval-print-Loop können im Gegensatz zu einem reinen Interpreter jedoch auch kompilierte Funktionen aufgerufen werden. Er macht die Sprache Lisp besonders geeignet für Rapid Prototyping.

Es wurde von uns zuerst ein Prototyp mit den gewünschten Features nur für binäre Constraints implementiert. Die Erweiterung auf beliebig-stellige Constraints verlief schnell und problemlos.

Lisp ist eine dynamisch getypte Sprache. Typen müssen nicht angegeben werden. Listen, als die wir Domains implementiert haben, können Elemente beliebig gemischten Typs aufnehmen. Das hat zur Folge, dass unser Solver nicht nur auf Ganzzahl-Domains arbeitet, sondern ebenso auf reellen Zahlen, Symbolen, etc., sofern die Closures der im CSP gespeicherten Constraints mit diesen Typen umgehen können.

Großen Vorteil brachten die weitgehenden objektorientierten Features von Common Lisp wie multiple Vererbung, Methodenkombination und die Möglichkeit, Methoden auf mehrere Klassen zu spezialisieren. Dies ermöglichte es uns, eine ganze Reihe von angepassten, teilweise kombinierten Strategien zu implementieren und gegeneinander zu testen.

Schließlich halfen uns Lisps Makros, unser Programm benutzerfreundlich zu gestalten, und den Implementierungsaufwand (etwa bei linearen Constraints) einzuschränken.

Der Compiler unserer Wahl war schlussendlich Clisp. Dieser ist mit Lisp in a Box im Paket mit dem Editor Emacs und der Common Lisp Entwicklungsumgebung Slime sowohl für Linux als auch für Windows kostenfrei erhältlich. Auch unter Unix sind Clisp und Slime verfügbar. Das gab für uns den Ausschlag für die Auswahl des Compilers, da es es uns ermöglichte, jeweils sowohl am Desktop-Rechner zu Hause, auf dem Laptop, als auch an der Universität zu unter den diversen Betriebssystemen zu arbeiten.

TUCS kompiliert jedoch auch mit dem stärker optimierenden SBCL (Steel Bank Common Lisp).

Zur Verwaltung unserer Dateien haben wir ein CVS genutzt.

11 Fazit

Unser Design hat sich als flexibel und erweiterbar erwiesen. Neue Strategien konnten mit minimalem Aufwand implementiert werden, was es uns auch ermöglichte, eine Vielzahl von Strategien in Kombination zu testen. Dies wurde uns durch die Implementierung der Strategie als Klasse ermöglicht, die auch die Verwaltung eigener Datenstrukturen erlaubt. Der durch diese starke Objektorientierung verursachte grundsätzliche Mehraufwand dürfte durch die Möglichkeit, leicht eine wesentlich effizientere Strategie zu implementieren, mehr als aufgewogen sein.

Die eigentliche Tiefensuche dagegen ist aus Effizienzgründen in Datenstrukturen und Methoden bewusst schlicht gehalten, wobei die effiziente Verarbeitung von Listen in Lisp genutzt wurde. Hierbei gingen wir anhand der Projektanforderung davon aus, dass wir nur mit für unsere Belange ausreichend kleinen Domains arbeiten.

Common Lisp hat sich für uns als sehr geeignete Wahl einer Implementierungssprache für eine möglichst allgemeingültige Formulierung eines diskreten, finiten CSP erwiesen.

12 Quellen und Links

http://en.wikipedia.org/wiki/Backjumping

http://en.wikipedia.org/wiki/Backmarking

Roman Barták: **On-Line Guide To Constraint Programming** Charles Universität Prag
http://ktiml.mff.cuni.cz/~bartak/constraints

Wolfgang Runte: **Ein hybrides Framework für Constraint-Solver.** Diplomarbeit Informatik an der Universität Bremen. 27. Januar 2006.
http://www.tzi.de/~woru/pub/diplom/html/Diplom.html

http://common-lisp.net/project/lispbox/

http://user.cs.tu-berlin.de/~langk/studium/TU/WS0506/typsysteme/lisp-clos.pdf