Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

2. Mai 2014

Konjunktive Normalform

- Ein *Literal* L ist eine Variable A oder eine negierte Variable $\neg A$

$$L = A \mid \neg A$$

  Wir bezeichnen Literale mit L, M oder N.

- Eine Klausel K ist eine Disjunktion von Literalen:

$$K = L_1 \vee L_2 \vee \cdots \vee L_m$$

- Eine Formel C ist in konjunktiver Normalform (KNF, CNF), wenn sie eine Konjunktion von Klauseln ist:

$$\begin{aligned} C &= K_1 \wedge K_2 \wedge \cdots \wedge K_n \\ &= (L_1 \vee \cdots \vee L_p) \wedge \cdots \wedge (M_1 \vee \cdots \vee M_q) \end{aligned}$$

Anwendung von CNF: Allgemeingültigkeit

- Überprüfen von Allgemeingültigkeit
- Eine Klausel ist allgemeingültig genau dann, wenn sie komplementäre Literale L_i, L_j enthält, d.h., $L_i = \neg L_j$ oder $L_j = \neg L_i$
- Eine Formel in CNF ist allgemeingültig, wenn jede ihrer Klauseln allgemeingültig ist
- Beispiel:

$$(L_1 \vee L_2 \vee \neg L_3 \vee \neg L_2) \wedge (L_2 \vee \neg L_3 \vee L_4 \vee L_5)$$

ist nicht allgemeingültig, da die zweite Klausel nicht allgemeingültig ist

Schreibweise

Formeln in CNF werden oft in Mengenschreibweise notiert. Statt

$$(L_1 \vee L_2 \vee \neg L_3 \vee \neg L_2) \wedge (L_2 \vee \neg L_3 \vee L_4 \vee L_5)$$

schreibt man dann

$$\{\{L_1, L_2, \neg L_3, \neg L_2\}, \{L_2, \neg L_3, L_4, L_5\}\}$$

oder auch nur

$$L_1, L_2, \neg L_3, \neg L_2 \qquad L_2, \neg L_3, L_4, L_5$$

Anwendung von CNF: Inkonsistenz

- Überprüfen von Inkonsistenz
- Eine Formel in CNF ist genau dann inkonsistent, wenn die leere Klausel ableitbar ist
- Beispiel:

$$\{\{L_1, L_2, \neg L_3, \neg L_2\}, \emptyset, \{L_2, \neg L_3, L_4, L_5\}\}$$

ist inkonsistent, da eine leere Klausel enthalten ist

Unit-Resolution

In den Übungen zum DPLL-Algorithmus wurde die folgende Ableitungsregel für Klauseln (Unit-Resolution) betrachtet: Wenn M und L_i komplementäre Literale sind, dann gilt

$$\frac{L_1, \ldots, L_m \qquad M}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m}$$

(Dass diese Regel korrekt ist überlegt man sich so: Da M und L_i komplementär sind kann man die linke Seite der Prämisse als

$$M \Rightarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$$

schreiben, und aus M und $M \Rightarrow \phi$ folgt ϕ.)

Beispiel

$L_1 \Rightarrow L_2$ und $L_2 \Rightarrow L_3$ haben die CNF $\neg L_1, L_2$ bzw. $\neg L_2, L_3$. Somit gilt die Ableitung

$$\dfrac{\dfrac{\neg L_1, L_2 \qquad L_1}{L_2} \qquad \neg L_2, L_3}{L_3}$$

Resolution

Die Beweisregel zur Unit-Resolution lässt sich folgendermaßen verallgemeinern: Falls L_i und M_j komplementär sind (d.h. $L_i \equiv \neg M_j$ oder $\neg L_i \equiv M_j$) gilt

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n}$$

Diese Form der Ableitung heißt (allgemeine) Resolution. $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n$ heißt *Resolvente*.

Die hier angegebene Resolutionsregel arbeitet immer auf Klauseln. Falls $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ keine komplementären Literale haben, so kann die Resolutionsregel nicht angewendet werden.

Resolution: Korrektheit

L_i, M_j komplementär ($L_i \equiv \neg M_j$ oder $\neg L_i \equiv M_j$):

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n}$$

Die Resolvente ist *nicht* äquivalent zu den Ausgangsklauseln. Sei z.B. η eine Belegung in der nur L_1 wahr ist. Dann ist die Resolvente wahr aber die Prämisse $M_1, \ldots, M_n$ falsch.

Aber: $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ sind nur dann beide erfüllbar, wenn die Resolvente erfüllbar ist; wenn also $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ beide wahr sind, so ist auch die Resolvente wahr.

(Das kann man sich so überlegen: Sei L_i wahr. Dann ist M_j falsch; damit $M_1, \ldots M_n$ gilt, musss also $M_1 \ldots, M_{j-1}, M_{j+1}, M_n$ gelten. Falls L_i falsch ist, kann man entsprechend mit $L_1, \ldots, L_m$ argumentieren.)

Resolution: Korrektheit

Seien C_1, C_2 Formeln in CNF. Dann haben C_1 und C_2 die Form $C_1 = K_1 \wedge \cdots \wedge K_m$, $C_2 = K'_1 \wedge \cdots \wedge K'_n$ mit Klauseln K_i, K'_j.

Wir sagen C_2 ist durch Resolution aus C_1 ableitbar und schreiben

$$C_1 \left|\frac{}{\text{res}}\right. C_2$$

wenn es für jede Klausel K'_j einen Ableitungsbaum gibt, in dem nur die Resolutionsregel verwendet wird, und an dessen Blättern nur die Klauseln $K_1, \ldots, K_m$ stehen.

Die Aussage auf der vorhergehenden Folie bedeutet, dass die Ableitungsbeziehung $C_1 \left|\frac{}{\text{res}}\right. C_2$ korrekt ist: aus wahren Prämissen kann durch Resolution keine falsche Aussage gezeigt werden.

Resolution: Vollständigkeit

Mit dem Sequenzenkalkül haben wir eine korrekte und vollständige Ableitungsregel kennengelernt:

$$\phi \models \psi \quad \text{genau dann, wenn} \quad \phi \left|\frac{}{\text{seq}}\right. \psi$$

Jede in einer „Wissensbasis“ ϕ wahre Formel ψ kann also mit dem Sequenzenkalkül aus ϕ abgeleitet werden (und es können keine falschen Formeln abgeleitet werden).

Gilt für den Resolutionskalkül eine ähnliche Eigenschaft? Man kann sich, z.B. fragen, ob jede wahre Formel C_2 in CNF aus einer Wissensbasis C_1, die nur aus Klauseln besteht, abgeleitet werden kann:

$$\text{Frage:} \quad C_1 \models C_2 \quad \text{genau dann, wenn} \quad C_1 \left|\frac{}{\text{res}}\right. C_2 \quad ?$$

Resolution

Resolution ist *nicht* vollständig: Sei C eine Formel in CNF (also eine Menge von Klauseln) und K eine Klausel. Aus $C \models K$ folgt nicht $C \not\vdash_{\text{res}} K$.

Beispiel: Da $A \vee \neg A$ allgemeingültig ist (also $\models A \vee \neg A$ wahr ist), gilt natürlich auch $B, \neg C, D \models A \vee \neg A$. Nachdem in $\{B, \neg C, D\}$ aber keine komplementären Literale vorkommen ist die Resolutionsregel gar nicht anwendbar, man kann also $A \vee \neg A$ nicht ableiten.

Resolution

Resolution ist aber *widerspruchsvollständig*: wenn C inkonsistent ist, dann lässt sich $\bot$ durch Resolution ableiten:

$$\text{Wenn} \quad C \models \bot \quad \text{dann} \quad C \left|\frac{}{\text{res}}\right. \bot$$

Das ist aber schon genug, um jeden aussagenlogischen Schluss (zwischen Formeln in CNF) durch Resolution beweisen zu können:

$$C_1 \models C_2 \quad \text{genau dann, wenn} \quad C_1 \wedge \neg C_2 \models \bot$$

(Die linke Seite sagt, dass in jeder Belegung, in der C_1 wahr ist auch C_2 wahr ist. Ist dass der Fall, so kann es keine Belegung geben, in der C_1 wahr ist und C_2 falsch; das ist genau die Aussage auf der rechten Seite. Offensichtlich gilt auch die umgekehrte Richtung.)

Resolutionsabschluss

Zum Beweis der Widerspruchsvollständigkeit führen wir folgenden Begriff ein:

Definition

Sei C eine Menge von Klauseln. Der Resolutionsabschluss $\mathrm{RC}(C)$ (resolution closure) von C ist der Fixpunkt der Iteration

$$C \leftarrow C \cup \mathfrak{R}(C),$$

wobei $\mathfrak{R}(C)$ die Menge aller Klauseln ist, die durch Anwendung der Resolutionsregel auf zwei Klauseln aus C entstehen.

Da C nur endlich viele Literale enthält und wir die Klauseln als Mengen betrachten ist $\mathrm{RC}(C)$ für jedes C definiert.

Widerspruchsvollständigkeit der Resolution

Theorem (Grundresolutionstheorem)

Wenn eine Menge von Klauseln C unerfüllbar ist, dann enthält der Resolutionsabschluss von C, $\mathrm{RC}(C)$ *die leere Klausel.*

Beweis: Falls $\mathrm{RC}(C)$ nicht die leere Klausel enthält können wir eine Belegung η konstruieren, die C erfüllt: Seien $L_1, \ldots, L_k$ die Literale aus C; sei $\eta(L_j)$ für alle $j < i$ definiert, dann definieren wir rekursiv $\eta(L_i)$ durch

- $\eta(L_j) =$ falsch, falls es eine Klausel in C gibt, die $\neg L_j$ enthält und deren andere Literale unter η alle falsch sind.
- Sonst ist $\eta(L_j)$ wahr.

Widerspruchsvollständigkeit der Resolution

Theorem (Grundresolutionstheorem)

Wenn eine Menge von Klauseln C unerfüllbar ist, dann enthält der Resolutionsabschluss von C, $\mathrm{RC}(C)$ die leere Klausel.

Beweis (Fortsetzung): $\eta(C) =$ wahr. Angenommen das wäre nicht der Fall, so gäbe es einen kleinsten Index i für den eine Klausel K aus $\mathrm{RC}(C)$ in der Konstruktion von η den Wert falsch erhielte. Dann hätte K die Form $L_1, \ldots, L_{i-1}, A_i$ oder $L_1, \ldots, L_{i-1}, \neg A_i$ mit $\eta(L_j) =$ falsch für $j < i$. Wäre nur eine der Klauseln in $\mathrm{RC}(C)$ so erhielte sie nach Konstruktion den Wert wahr; es müssten also beide Klauseln in $\mathrm{RC}(C)$ sein. Dann wäre aber die Resolvente der beiden Klauseln ebenfalls in $\mathrm{RC}(C)$ und falsch, im Widerspruch zur Minimalität von K. $\square$

Theorembeweisen durch Resolution

Beim Theorembeweisen durch Resolution geht man folgendermaßen vor:

Gezeigt werden soll $\phi \models \psi$.

- Wandle $\phi \wedge \neg\psi$ in CNF um
- Versuche daraus $\bot$ (die leere Klausel) durch Resolution abzuleiten

Gelingt das, ist $\phi \wedge \neg\psi$ widersprüchlich, somit gilt $\phi \models \psi$.

Gelingt das nicht ist $\phi \wedge \neg\psi$ konsistent, $\phi \models \psi$ gilt damit nicht. Aus dem Beweis des Grundresolutionstheorems lässt sich eine Belegung η konstruieren, die ϕ und $\neg\psi$ erfüllt.

SAT

Gegeben sei eine aussagenlogische Formel ϕ. Das *Erfüllbarkeitsproblem (SAT)* ist die Frage nach einer erfüllenden Belegung: Gibt es ein η mit

$$\models_\eta \phi?$$

SAT war das erste Problem, das als *NP-vollständig* bewiesen wurde.

Praktische Lösungsverfahren

Das bisher betrachtete Verfahren *SAT*-Probleme mit Wahrheitstabellen zu lösen ist nur für sehr kleine Probleme praktisch durchführbar.

Obwohl es aufgrund der NP-Vollständigkeit von *SAT* (wahrscheinlich) kein effizientes Verfahren gibt um *alle SAT* Probleme zu lösen, gibt es Verfahren, die *fast alle SAT*-Probleme viel effizienter lösen als die Berechnung der vollständigen Wahrheitstabelle.

Verfahren wie DPLL oder WalkSat können fast alle (und insbesondere fast alle praktisch auftretenden) *SAT*-Probleme gut lösen. Nur Probleme mit einer besonderen Struktur bzw. einem bestimmten Verhältnis von Klauseln zu Variablen sind für diese Algorithmen problematisch. Im Folgenden werden wir einige der Techniken betrachten, die dafür eingesetzt werden.

DPLL-Algorithmus

Das DPLL-Verfahren zur Lösung von *SAT*-Problemen baut auf folgenden Techniken auf:

- Basis: Backtracking-Algorithmus
- Effizientere Implementierung: „klassisches“ DPLL
  - Frühzeitiger Abbruch
  - Elimination von reinen Literalen (pure symbol heuristic, pure literal elimination)
  - Unit-Propagation (unit clause heuristic)
- Noch Effizientere Implementierung: „modernes“ DPLL
  - Iterativer Algorithmus
  - Heuristik zur Variablenauswahl
  - Conflict-directed Backjumping
  - Lernen von Klauseln
  - Beobachtete Literale

SAT durch Wahrheitstabellen

Eine Formel ϕ ist *erfüllbar* (satisfiable), wenn sie für mindestens eine Belegung wahr ist, wenn also gilt

$$\exists \eta : \mathcal{A} \to \mathbb{B} : [\![\phi]\!]\eta = \text{wahr}$$

Eine Formel ϕ ist genau dann erfüllbar, wenn in der letzten Spalte ihrer Wahrheitstabelle mindestens einmal der Wert wahr vorkommt.

$\cdots$	ϕ
$\cdots$	$\vdots$
$\cdots$	wahr
$\cdots$	$\vdots$

Backtracking-Suche

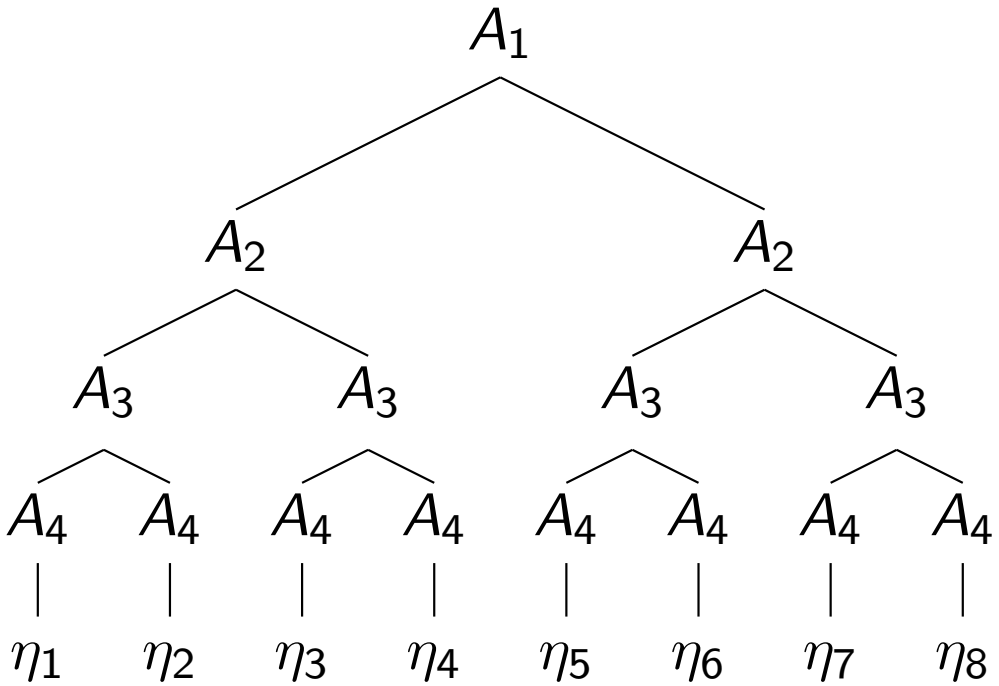
- Zustandsraum: alle möglichen (partiellen) Belegungen der Variablen
- Rekursive Aufzählung des Zustandsraumes
- Faktorisierte Darstellung der Zustände:
  $\{A_1 = \eta_1, \ldots, A_k = \eta_k\}$
- Suchoperatoren: Hinzufügen einer neuen Variablenbelegung

$$\{A_1 = \eta_1, \ldots, A_k = \eta_k\} \rightarrow \{A_1 = \eta_1, \ldots, A_k = \eta_k, A_{k+1} = \eta_{k+1}\}$$

- Feste Ordnung der Variablen

Backtracking-Suche

Bei der „naiven“ Backtracking-Suche wird der komplette Zustandsraum durchlaufen:



Implementierung

Siehe die Datei `boolean.ts` für eine TypeScript (JavaScript) Implementierung.

Verbesserung: Frühzeitiger Abbruch der Suche

Die erste Implementierung der Backtracking-Suche (`ttTautology` für Allgemeingültigkeit und ttSatisfiable für Erfüllbarkeit) erzeugt immer eine vollständige Belegung der Variablen und wertet dann den Term aus.

In vielen Fällen kann man die Suche schon früher abbrechen:

$$A \vee \neg A \vee (B \wedge C) \vee (A \wedge B)$$
$$(A \vee B) \wedge (\neg A \wedge C \wedge D) \wedge \neg B$$

Beim ersten Term ist schon nach der Belegung von A durch einen beliebigen Wert klar, dass der Term wahr ist; beim zweiten Term kann man nach der Wahl von A und B schon feststellen, dass kein Wert von C oder D den Term noch erfüllen kann.

Auswertung bei partieller Belegung

Um mit partiellen Variablenbelegungen umgehen zu können kann man die Wahrheitstabellen folgendermaßen erweitern:

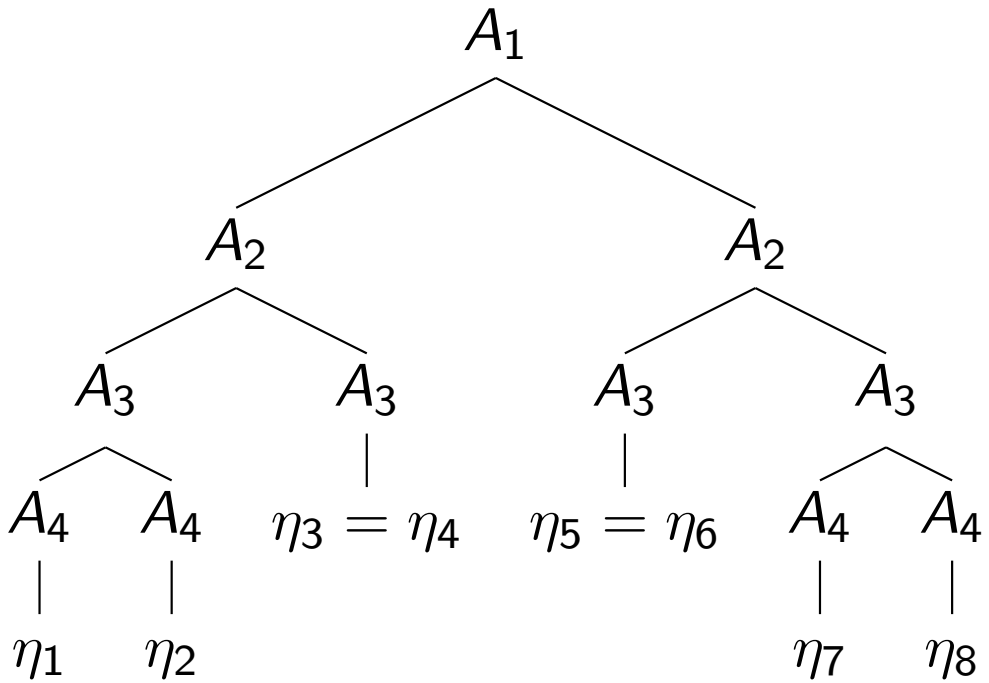
	$\neg$
falsch	wahr
wahr	falsch
undef	undef

$\wedge$	falsch	wahr	undef
falsch	falsch	falsch	falsch
wahr	falsch	wahr	undef
undef	falsch	undef	undef

$\vee$	falsch	wahr	undef
falsch	falsch	wahr	undef
wahr	wahr	wahr	wahr
undef	undef	wahr	undef

Backtracking-Suche

Damit kann man in vielen Fällen die Suche schon früher abbrechen:



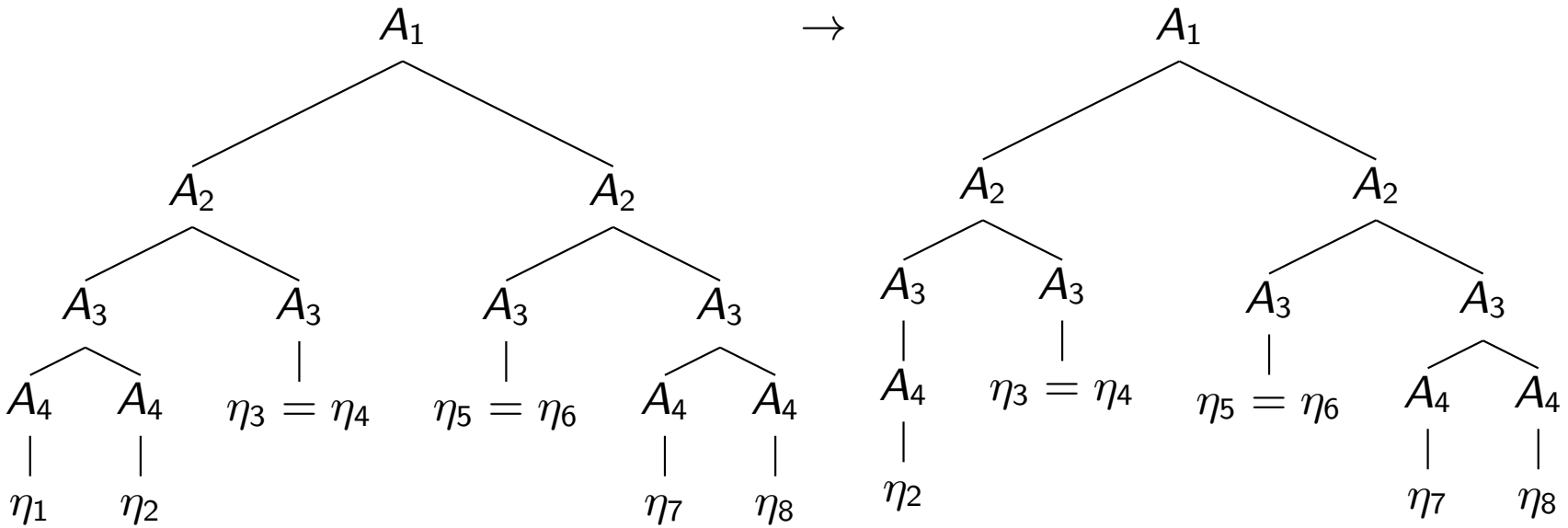
Das ist in der Funktion `dpll1` implementiert, die den Wahrheitswert von Termen mit `evaluate3` auswertet.

Verwendung von CNF

- Der bisherige Algorithmus ist für beliebige Formeln verwendbar
- Die Auswertung ist immer noch nicht effizient
- Für weitere Verbesserungen benötigen wir mehr Information über die Struktur der Terme
- Diese Struktur bekommen wir z.B. durch CNF
- Keine Änderung am Algorithmus *nötig*, sofern wir in CNF das Interface implementieren, das zum Backtracking benötigt wird
- Viele Verbesserungen *möglich*, wenn wir auf CNF umsteigen

Propagation

Bei der Propagation wird der Suchraum verkleinert, indem wir die Struktur des Problems oder der Lösung verwenden, um gewisse Zweige des Suchbaums auszuschließen:



Propagation

Beispiel: Wenn eine Klausel nur ein Literal L enthält, so muss die entsprechende Variable A `true` oder `false` sein, je nachdem ob L positiv (nicht negiert, d.h. $L \equiv A$) oder negativ (negiert, d.h. $L \equiv \neg A$) ist.

Ein solches Literal kann man dann aus anderen Klauseln entfernen; das entspricht genau der Anwendung der Unit-Propagationsregel.

DPLL-Algorithmus

- Basis: Backtracking-Algorithmus
- Effizientere Implementierung
  - Frühzeitiger Abbruch
  - Elimination von reinen Literalen (pure symbol heuristic, pure literal elimination)
  - Unit-Propagation (unit clause heuristic)
- Noch Effizientere Implementierung
  - Iterativer Algorithmus
  - Heuristik zur Variablenauswahl
  - Conflict-directed Backjumping
  - Lernen von Klauseln
  - Beobachtete Literale

Die in blau markierten Techniken lassen sich auch für allgemeinere Probleme einsetzen. Wir werden sie deshalb im Rahmen des Constraint-Processing Frameworks erläutern.

Constraint Processing

Ein Constraint beschreibt (je nach Sichtweise)

- die Einschränkung von Alternativen oder
- das Offenhalten von Möglichkeiten

Ein Constraintproblem charakterisiert eine Situation durch eine Menge von Constraints. Constraint Processing bietet automatische Lösungsverfahren für Constraintprobleme an.

Beispiele für Constraintprobleme

- Planung von Funknetzwerken
  - Frequenzzuweisung
  - Aufstellung von Sendemasten
- Planung von Produktionsprozessen (job-shop scheduling)
- Layout von Elektronischen Bauteilen
- Diagnose von Fehlern in Schaltungen
- Erkennen von dreidimensionalen Szenen (Huffman-Clowes labeling)
- Kartenfärbung/Graphenfärbung (graph-coloring problems)

Relationen/Constraints

Definition

Sei $X = \{x_1, \ldots, x_n\}$ eine Menge von Variablen mit Domänen $D = \{D_1, \ldots, D_n\}$.

Eine Relation R über $S \subseteq X$ ist eine Teilmenge von $\prod_{x_s \in S} D_s$; S heißt *Scope* von R (scope(R)).

Ein Constraint ist eine Relation.

Beispiel

Seien $X = \{x1, x2\}$, $D_1 = \{R, G, B\}$, $D_2 = \{\text{Auto}, \text{Blume}\}$. Dann sind

$$C_1 = \emptyset$$
$$C_2 = \{\langle R, \text{Auto}\rangle, \langle G, \text{Auto}\rangle, \langle R, \text{Blume}\rangle\}$$
$$C_3 = D_1 \times D_2$$

Constraints mit Scope $\{x_1, x_2\}$.

$\{\langle R\rangle, \langle G\rangle\}$ ist ein Constraint mit Scope $\{x_1\}$

Relationale Algebra

Seien R, R' Relationen mit demselben Scope. Dann ist

- $R \cup R'$ die Relation, die alle Tupel aus R und R' enthält
- $R \cap R'$ die Relation, die nur die Tupel enthält, die in R und R' vorkommen und
- $R \setminus R'$ oder $R - R'$ die Relation, die nur die Tupel aus R enthält, die nicht in R' vorkommen

Relationale Algebra

Seien R und R' Relationen mit beliebigen Scopes S und S', und sei $Y \subseteq X$. Dann ist

- $\sigma_{x_1=a_1,\ldots,x_n=a_n}(R)$ (Selektion) die Menge aller Tupel aus R die an der Stelle x_i den Wert a_i haben.
- $\pi_Y(R)$ (Projektion) die Menge aller Tupel aus R beschränkt auf Variablen in Y.
- $R \bowtie R'$ (Join) eine Relation mit Scope $S \cup S'$, deren Elemente alle Tupel t sind, für die $\pi_S(t) \in R$ und $\pi_{S'}(t) \in R'$ gilt.

Graphen

Definition

Ein (ungerichteter) Graph $G = (V, E)$ besteht aus einer endlichen Menge V von Knoten (vertices, nodes), und einer Menge E von Kanten (edges, arcs). Jede Kante ist eine (ungeordnete) 2-Menge $\{u, v\}$ von Knoten; Kanten werden oft in der Form (u, v) geschrieben. Eine Kante (u, v) *verbindet* u und v, man sagt dann u und v sind *benachbart* oder *adjazent*. Die Anzahl der Nachbarn eines Knotens heißt sein *Grad*.

Ein *Pfad* ist eine Folge von Kanten $e_1, \ldots, e_k$, so dass e_i und e_j einen gemeinsamen Knoten haben. Man schreibt einen Pfad auch als Folge der Knoten $v_1, \ldots, v_{k+1}$. Ein Pfad ist *einfach*, wenn kein Knoten doppelt darin vorkommt. Ein Graph ist zusammenhängend, wenn zwei beliebigen Knoten durch einen Pfad verbunden sind.

Graphen

Definition

Ein *Kreis* ist ein Pfad, dessen Anfangs- und Endpunkt übereinstimmen. Ein Kreis ist *einfach*, wenn nur der Start- und Endpunkt doppelt vorkommen. Ein ungerichteter Graph ohne Kreise ist ein *Baum*. Ein Graph ist vollständig, wenn zwei beliebige Knoten benachbart sind.

Constraint-Netzwerke

Definition

Ein Constraint-Netzwerk $\mathfrak{R} = (X, D, C)$ besteht aus einer endlichen Menge $X = \{x_1, \ldots, x_n\}$ von Variablen mit Domänen $D = \{D_1, \ldots, D_n\}$ und einer Menge von Constraints $C = \{C_1, \ldots, C_k\}$. Die Menge aller Scopes von Constraints heißt das *Schema* des Constraint-Netzwerks.

Ein Constraint beschreibt, welche Werte für die Variablen aus seinem Scope gleichzeitig zulässig sind.

Beispiel: Karte von Australien

Beispiel: Karte von Australien

- Färben der Karte durch drei Farben: R, G, B
- Benachbarte Regionen dürfen nicht die gleiche Farbe bekommen

Beispiel: Karte von Australien

- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{R, G, B\}$
- $C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW,$
  $SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
- Alle Constraints haben 2 Variablen

Beispiel: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Sudoku

- 81 Quadrate: $A1$–$I9$
- Jedes soll mit Ziffern 1–9 belegt werden
- Einige Quadrate schon ausgefüllt
- Keine Ziffer kommt zweimal in einer Zeile, Spalte oder 3×3 Box vor

Sudoku Constraints

$\text{AllDifferent}(A1, A2, \ldots, A9)$

$\text{AllDifferent}(B1, B2, \ldots, B9)$

$\ldots$

$\text{AllDifferent}(A1, B1, \ldots, I1)$

$\text{AllDifferent}(A2, B2, \ldots, I2)$

$\ldots$

$\text{AllDifferent}(A1, A2, A3, B1, B2, B3, C1, C2, C3)$

$\text{AllDifferent}(A4, A5, A6, B4, B5, B6, C4, C5, C6)$

$\ldots$

Sudoku

Sudoku ist mit *globalen Constraints* definiert. Es kann aber in ein binäres Problem (mit erheblich mehr Constraints) umgewandelt werden:

$$\begin{aligned}\text{AllDifferent}(A1, A2, \ldots, A9) = {} & A1 \neq A2 \wedge \\ & A1 \neq A3 \wedge \\ & A1 \neq A4 \wedge \\ & \ldots \\ & A2 \neq A3 \wedge \\ & \ldots\end{aligned}$$

Graph eines Constraint Netzwerks

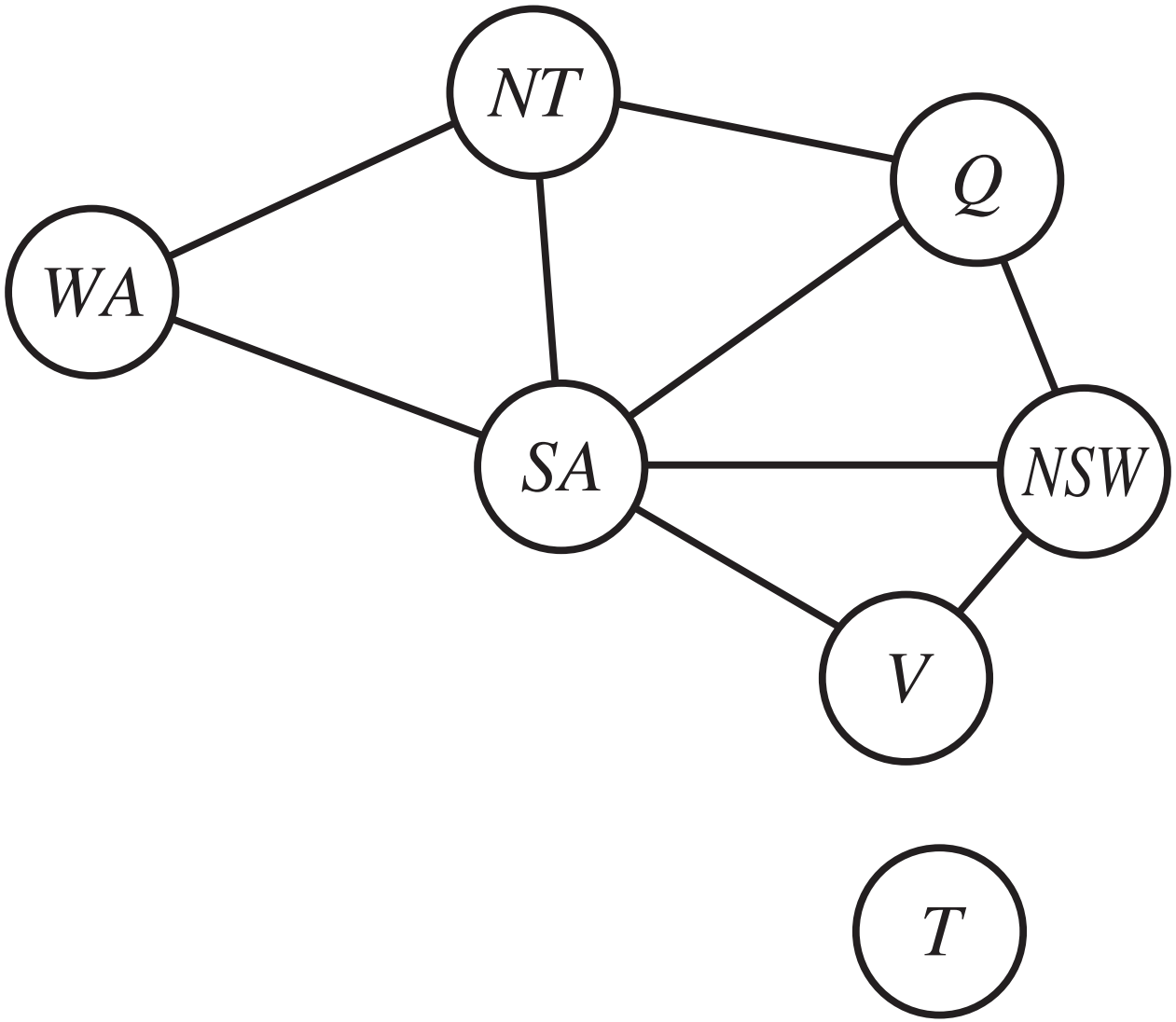
Ein Constraint-Netzwerk wird häufig durch seinen Graphen veranschaulicht.

- Die Knoten des Graphen sind die Variablen
- Zwei Knoten sind durch eine Kante verbunden, wenn es einen Constraint gibt, in dessen Scope beide Variablen vorkommen.

Beispiel: Karte von Australien

- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{R, G, B\}$
- $C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW,$
  $SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
- Alle Constraints haben 2 Variablen

Beispiel: Karte von Australien



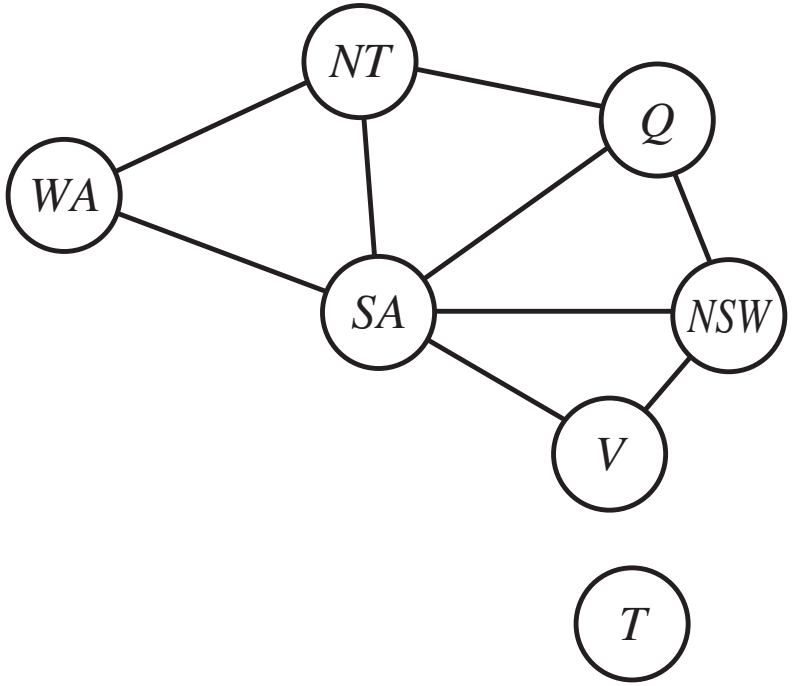
Instanziierungen

Definition

Eine Instanziierung einer Teilmenge $Y \subseteq X$ von Variablen ist eine Zuweisung, die jeder Variable aus Y ein Element ihrer Domäne zuweist. Man kann eine Instanziierung also als eine Relation mit Scope Y auffassen, die nur ein einziges Tupel enthält. Eine Instanziierung I erfüllt einen Constraint C_k, wenn $\pi_{\text{scope}(C)}(I) \in C_k$ ist, wenn also $\text{scope}(Y) \supseteq \text{scope}(X)$ ist und alle Werte von I zusammen in C_k vorkommen dürfen.

Beispiel: $(x_1 = R)$ ist eine Instanziierung für die vorhin angegebene Relation.

Beispiel



- $(WA = R, NT = R, Q = R, NSW = R)$ ist eine Instanziierung, die keinen Constraint erfüllt
- $(WA = R, NT = G, Q = B, SA = R)$ ist eine Instanziierung, die die Constraints $WA \neq NT$, $NT \neq Q$ und $SA \neq Q$ erfüllt, aber nicht $WA \neq SA$

Konsistente Instanziierungen, Lösungen

Definition

Eine partielle Instanziierung mit Scope Y ist *konsistent*, wenn sie alle Constraints erfüllt, deren Scopes Teilmengen von Y sind.

Eine Lösung eines Constraint-Netzwerks (X, DC) ist eine konsistente Instanziierung mit Scope X.

Beispiel: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Das rechte Spielfeld ist eine Lösung für das Sudoku Netzwerk

Propagation

Die Betrachtung von Constraints zwischen Variablen ermöglicht es, das Constraintsystem zu „verbessern“ indem man Werte, die mit Constraints in Widerspruch stehen aus den Domänen der Variablen entfernt. Das wird als „lokale Konsistenz“ bezeichnet.

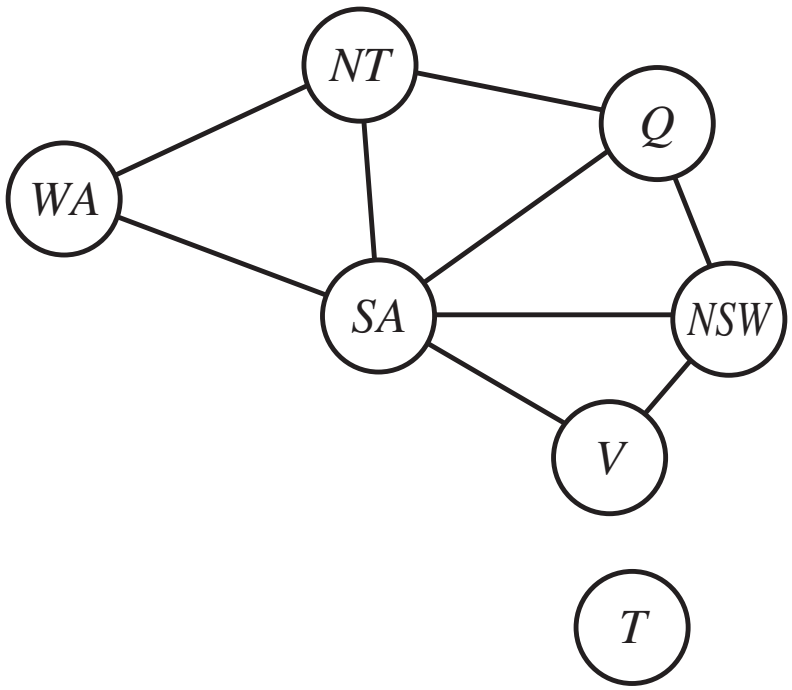
Es gibt verschiedene Formen von lokaler Konsistenz:

- Knotenkonsistenz, 1-Konsistenz (node consistency): nur Werte, die mit den unären Constraints vereinbar sind werden in den Domains behalten
- Bogenkonsistenz, 2-Konsistenz (arc consistency): Sei x_i eine Variable und x_j eine benachbarte Variable (d.h., es gibt einen Constraint mit Scope $\{x_i, x_j\}$. Zu jedem Wert von x_i gibt es einen Wert im Bereich von x_j, der konsistent ist.
- Pfadkonsistenz, 3-Konsistenz (path consistency): Zu jeder konsistenten Instanziierung von x_i, x_j und jedem zu x_i, x_j benachbarten Knotens x_k gibt es eine konsistente Instanziierung von x_i, x_j, x_k.

(Starke) k-Konsistenz

- Ein CSP ist k-konsistent, wenn sich jede konsistente Instanziierung von $k-1$ Variablen zu jeder anderen Variablen fortsetzen lässt.
- Ein CSP ist stark k-konsistent, wenn es 1, 2, ..., k-konsistent ist.

Beispiel Knotenkonsistenz



Beispiel: Bogenkonsistenz

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Das rechte Spielfeld ist eine Lösung für das Sudoku Netzwerk

Beispiel Pfadkonsistenz