Lehrstuhl für Informatik I
Informationsstrukturen und wissensbasierte Systeme
Universität Würzburg

Prof. Dr. H. Noltemeier
Dr. S. O. Krumke

Würzburg, den 26. November 1997

Informationsstrukturen und wissensbasierte Systeme
(WS 97/98)

1. Fallstudie

1 Generelles

Im Rahmen der Fallstudien sollen ausgewählte Algorithmen von den einzelnen Arbeitsgruppen implementiert und experimentell untersucht werden. Jede Arbeitsgruppe besteht aus maximal zwei Studenten. Die von den Gruppen entwickelten Programme werden von den studentischen Hilfskräften abgenommen. Für jedes Projekt gibt es konkrete Instanzen der jeweiligen Probleme, die bei der Abnahme – neben anderen Beispielen – vorgeführt werden sollen.

Von jeder Gruppe wird ferner eine schriftliche Ausarbeitung erwartet. In dieser Ausarbeitung sollen kurz die Aufgabenstellung, notwendige Definitionen und die Ergebnisse zusammengefaßt werden.

1.1 Rechnerzeiten im CIP-Pool

Für die Programmierung stehen die CIP-Pools SE05 und SE06 mittwochs von 8 bis 12 Uhr zur Verfügung. Zu diesen Zeiten sind auch die studentischen Hilfskräfte anwesend, um Fragen zu beantworten.

1.2 Zeitplan

Die Abnahme der Programme erfolgt durch die studentischen Hilfskräfte im CIP-Pool jeweils zu den Rechnerzeiten bis zum 15. Januar 1998. Die schriftlichen Ausarbeitungen sind bis zum gleichen Termin ebenfalls bei den studentischen Hilfskräften abzugeben.

1.3 Richtlinien für die Programme

1. Implementierungssprache für die Programme ist C++. Für die Programmierung steht der GNU C++-Compiler zur Verfügung.
2. Es wird übersichtlicher und gut dokumentierter Code erwartet.
3. Zum Programm ist ein „Makefile“ zu erzeugen, mit dem das Compilieren/Linken erfolgt.

2 Projekte

2.1 Algorithmen für 3Sat

2.1.1 Übersicht

Ziel des Projektes ist die Implementierung und experimentelle Analyse von effizienten Algorithmen zum Lösen des Erfüllbarkeitsproblems 3-SAT.

Ein „schnelles“ Verfahren für 3-SAT soll mit klassischen Lösungsansätzen wie dem Evaluieren der Wahrheitstabelle und der aussagenlogischen Resolution verglichen werden.

Literatur: [3–5]

2.1.2 Kurze Darstellung der Grundbegriffe

Es sei $X = \{A_1, \ldots, A_n\}$ eine Menge von atomaren Formeln. Man nennt A_i und $\neg A_i$ *Literale*. Das Literal A_i ist *positiv*, das Literal $\neg A_i$ *negativ*. Im folgenden steht $\neg L$ für das negierte Literal, also $\neg A_i$, falls $L = A_i$ bzw. A_i falls $L = \neg A_i$.

Eine Instanz (X, M) des *Erfüllbarkeitsproblems* SAT ist durch eine Menge X von Atomformeln und eine Menge M von Klauseln über X gegeben. Das Problem besteht darin zu entscheiden, ob M erfüllbar ist. Mit k-SAT bezeichnen wir die Teilmenge der Instanzen von SAT, in der jede Klausel *genau* k Literale besitzt.

Seien C_1 und C_2 Klauseln. Dann heißt R *Resolvent* von C_1 und C_2, wenn es ein Literal L gibt mit $L \in C_1$ und $\neg L \in C_2$ und R die Form $R = (C_1 \setminus \{L\}) \cup (C_2 \setminus \{\neg L\})$ besitzt. Falls R die leere Menge ist, so wird diese dann auch durch $R = \Box$ bezeichnet ($\Box$: “*leere Klausel*”).

Sei M eine Klauselmenge. Dann ist $\mathrm{Res}(M)$ definiert durch:

$$\mathrm{Res}(M) := M \cup \{\, R : R \text{ ist Resolvent zewier Klauseln in } M \,\}.$$

Ferner definiert man

$$\begin{aligned} \mathrm{Res}^0(M) &:= M \\ \mathrm{Res}^{k+1}(M) &:= \mathrm{Res}(\mathrm{Res}^k(M)) \quad \text{für } k \geq 0 \end{aligned}$$

sowie

$$\mathrm{Res}^*(M) := \bigcup_{k \geq 0} \mathrm{Res}^k(M).$$

Man kann leicht zeigen, daß es für jede endliche Klauselmenge F ein $r \geq 0$ gibt, so daß

$$\mathrm{Res}^r(M) = \mathrm{Res}^{r+1}(M) = \cdots = \mathrm{Res}^*(M)$$

gilt. Nach dem *Resolutionssatz* der Aussagenlogik (siehe etwa [5, Kapitel 1]) ist die Klauselmenge M genau dann unerfüllbar, wenn $\Box \in \mathrm{Res}^*(M)$ gilt.

2.1.3 Algorithmen

Algorithmus für 3-Sat Schwerpunkt ist der „schnelle“ Algorithmus für 3-SAT aus dem Artikel [4]. Dieser Algorithmus besitzt eine Laufzeit von $\mathcal{O}(1.415^n)$, wobei n die Anzahl der Atomformeln in einer Instanz von 3-SAT bezeichnet.

Aussagenlogische Resolution Das Verfahren der Resolution (siehe etwa [5, Kapitel 1]) beruht auf dem Resolutionssatz der Aussagenlogik. Das Verfahren berechnet die *Resolventenhülle* $\mathrm{Res}^*(M)$ und prüft, ob $\Box \in \mathrm{Res}^*(M)$ gilt.

2.1.4 Aufgaben

Die zwei oben genannten Verfahren sollen implementiert werden. Durch experimentelle Analyse soll die Laufzeit des schnellen Algorithmus für 3-SAT mit der Resolution und dem „brute-force“ Ansatz über die Wahrheitstabelle verglichen werden.

2.1.5 Eingabespezifikation

Eine Instanz von 3-SAT soll als ASCII-Datei in folgendem Format eingegeben werden:

Eine Klausel $C = \{L_1, \dots, L_3\}$ werde als Zeile von ganzen Zahlen dargestellt, wobei für die $i$te Zahl z_i gilt:

$$z_i = \begin{cases} j & \text{falls } L_j = A_j \text{ mit der Atomformel } A_j \\ -j & \text{falls } L_j = \neq A_j \text{ mit der Atomformel } A_j. \end{cases}$$

Beispielsweise werde $C = \{A_1, \neg A_4, A_5\}$ durch die Zahlenfolge 1, -4, 5 repräsentiert. Die Zahlen sind dabei durch eine beliebige Anzahl von „Whitespaces“ getrennt.

Die Eingabedatei soll zunächst die Anzahl n der Atomformeln, dann die Anzahl m der Klauseln und letztendlich die Codierungen der einzelnen Klauseln enthalten:

n `<Zeilenumbruch>`
m `<Zeilenumbruch>`
C_1 `<Zeilenumbruch>`
$\vdots$
C_m

2.1.6 Aussagespezifikation

Die Algorithmen sollen jeweils nach `stderr` ausgeben, ob die Klauselmenge in der Instanz erfüllbar ist oder nicht. Zusätzlich ist als Rückgabewert des Programs der Wert 1 bei Erfüllbarkeit und 0 bei Unerfüllbarkeit zu liefern.

2.2 Algorithmen für quantitative temporale Constraint-Netze

2.2.1 Übersicht

In diesem Projekt sollen verschiedene Algorithmen für quantitative Constraint Satisfaction Probleme, die Konsistenz oder eine Approximation daran im Constraintgraphen erzeugen, implementiert und verglichen werden. Hauptfrage ist, wie effizient die einzelnen Algorithmen Inkonsistenzen entdecken.

Literatur: [1, 2, 6]

2.2.2 Kurze Darstellung der Grundbegriffe

Eine Instanz eines quantitativen temporalen Constraint Satisfaction Problems (TCSP) [2] ist durch eine Menge $X_1, \dots, X_n$ von Variablen und eine Menge $\mathcal{C}$ von Constraints gegeben. Jede Variable X_i besitzt einen kontinuierlichen Grundbereich. Jedes Constraint $C \in \mathcal{C}$ ist eine Menge $C = \{I_1, \dots, I_r\}$ von Intervallen $I_j = [a_j, b_j]$, $j = 1, \dots, r$. Ein unäres Constraint C_i beschränkt den Bereich der Variablen X_i auf die Werte in den entsprechenden Intervallen. Ein binäres Constraint C_{ij} beschränkt den Wert von $X_j - X_i$ auf die entsprechenden Intervalle.

Ein Tupel $X = (x_1, \ldots, x_n)$ heißt eine Lösung der Instanz, wenn die Zuweisung $X_1 := x_1, \ldots, X_n := x_n$ der Werte zu den Variablen alle Constraints erfüllt. Eine Instanz des TCSP ist *konsistent*, wenn es mindestens eine Lösung besitzt. Das Problem, die Konsistenz von TCSP zu entscheiden ist NP-vollständig. Daher ist man an Algorithmen interessiert, die manche, aber nicht alle Inkonsistenzen entdecken.

Ein TCSP kann durch einen gerichteten Graphen $G = (V, R)$ dargestellt werden. Die Eckenmenge V entspricht der Variablenmenge X und einer zusätzlichen Variablen X_0, die den *Startzeitpunkt des Universums* angibt. Die Verwendung von X_0 ermöglicht die Beschränkung auf binäre Constraints.

Ferner ist genau dann $(X_i, X_j) \in R$, wenn ein Constraint C_{ij} spezifiziert ist. Die gerichtete Kante (X_i, X_j) ist mit den Intervallen des Constraints C_{ij} markiert.

2.2.3 Algorithmen

Algorithmus „Path Consistency" (PC) Der Algorithmus PC [2, 6] stellt in einem TCSP Pfad-Konsistenz her. Dieser Algorithmus besitzt im Gegensatz zu den weiteren Algorithmen des Projekts keine polynomiale Zeitkomplexität.

Simple Temporal Problems Simple Temporal Problems (STP) sind TCSPs, bei denen jedes Constraint nur genau ein Intervall enthält. Bei diesen Problemen läßt sich Pfad-Konsistenz effizient durch durch Berechnung kürzester Wege erreichen. Die Konsistenz kann durch Prüfen auf negative Kreise entschieden werden. Klassische Algorithmen für kürzeste Wege und negative Kreise findet man etwa in [1].

Algorithmus „Upper-Lower-Tightening" (ULT) und Varianten Diese Algorithmen [6] erzeugen etwas „gröbere" Constraints als PC. Die Motivation ist dabei, die sogenannte Fragmentierung, die bei PC auftreten kann, zu vermeiden. Varianten von ULT sind DULT und ULT-2.

Algorithmus „Loose Path Consistency" (LPC) Dieser Algorithmus [6] steht „zwischen" PC und ULT. Er erzeugt schärfere Constraints, liefert im allgemeinen aber nicht die Resultate von PC. Von LPC existieren wieder Variaten DLPC und LPC-2.

2.2.4 Aufgaben

Die oben genanten Verfahren sollen implementiert werden. Anhand der Implementierungen sollen folgende Fragen sowohl für kontinuierliche als auch für ganzzahlige Grundbereiche untersucht werden:

1. Welcher der polynomialen Algorithmen entdeckt Inkonsistenzen am besten?

   Man kann zeigen, daß jeder der oben aufgeführten Algorithmen korrekt in dem Sinne ist, daß er nur dann Inkonsistenz meldet, wenn auch wirklich Inkonsistenz des Netzwerkes vorliegt. Da die Entscheidung der Konsistenz wie oben erwähnt NP-vollständig ist, sollen dabei die Algorithmen *relativ* untereinander verglichen werden.

2. Bei welchem Algorithmus ist das Verhältnis von Laufzeit zur Effektivität am besten?

   Es soll die Effektivität, gemessen an der Anzahl der entdeckten Inkonsistenzen, der Laufzeit gegenübergestellt werden.

Als Parameter für die Testinstanzen sind neben der *Größe* der Instanzen ihre *Schärfe* und die *Anzahl* der Constraints zu variieren (cf. [6]).

2.2.5 Eingabespezifikation

Eine Instanz eines TCSP soll als bewerteter Graph in einer ASCII-Datei in folgendem Format eingegeben werden:

Ein Constraint $C_{ij} = \{[a_1, b_1], \ldots, [a_r, b_r]\}$ wird als Zeile

$$i \quad j : r \quad a_1 \quad b_1 \cdots a_r \quad b_r$$

dargestellt. Dabei soll davon ausgegangen werden, daß alle Constraints binär sind.

Die Eingabedatei soll zunächst die Anzahl n der Variablen, dann die Anzahl m der Constraints enthalten. Danach folgt die Spezifikation der Constraints im oben genannten Format.

2.2.6 Aussagespezifikation

Die Algorithmen sollen Informationen über die Konsistenz der Instanz als Text nach `stderr` ausgeben. Zusätzlich ist als Rückgabewert des Programs der Wert 1 bei entdeckter Inkonsistenz und 0 sonst zu liefern. Der bei Abbruch entstandene Constraintgraph soll im Format der Eingabe nach `stdout` ausgegeben werden, so daß eine I/O-Umlenkung in eine Datei möglich ist.

2.3 Binäre Constraint Satisfaction Probleme

2.3.1 Übersicht

Ziel des Projektes ist es, verschiedene Algorithmen zum Erreichen von Konsistenz in Constraint Satisfaction Problemen auf ihre Effizienz beim Preprocessing-Einsatz für Backtracking zu untersuchen.

Literatur: [7, 8]

2.3.2 Kurze Darstellung der Grundbegriffe

Eine Instanz eines Constraint Satisfaction Problems (CSP) ist durch ein Tripel (Z, D, C) gegeben, wobei

1. $Z = \{X_1, \ldots, X_n\}$ eine endliche Menge von Variablen ist,
2. D eine Funktion, die jeder Variablen aus Z einen endlichen Grundbereich zuweist, und
3. C eine endliche Menge von Constraints ist.

   Ein *Compound Label* $(\langle X_1, v_1\rangle, \ldots, \langle X_k, v_k\rangle)$ ist eine (simultane) Zuweisung von Werten v_i zu Variablen X_i, $i = 1, \ldots, k$. Ein *Constraint* ist konzeptionell eine Menge von Compound Labels. Für $S \subseteq Z$ benutzt man C_S für das Constraint auf den Variablen aus S. Ein Constraint C_S heißt *binär*, wenn $|S| = 2$. Ein *binäres CSP* enthält nur binäre Constraints. Ein binäres CSP induziert einen Graphen mit Eckenmenge Z, in dem es eine (ungerichtete) Kante zwischen X_i und X_j gibt, wenn ein Constraint C_{X_1, X_j} spezifiziert ist.

Das CSP besteht darin, ein Compound Label zu finden, das alle Constraints erfüllt. Eine Instanz des CSP ist *erfüllbar*, wenn es mindestens ein solches Compound Label gibt.

Man nennt eine Instanz eines CSP *knotenkonsistent* (oder 1-*konsistent*), wenn für alle Variablen alle Werte aus dem Grundbereich die Constraints erfüllen. Eine Kante (X, Y) im Constraintgraphen ist *kantenkonsistent*, wenn es für jeden Wert v_x aus dem Grundbereich von X, der C_X erfüllt, einen Wert v_y aus dem Grundbereich von Y gibt, so daß das Compound Label $(\langle X, v_x \rangle, \langle Y, v_y \rangle)$ die Constraints C_X, C_Y und $C_{X,Y}$ erfüllt. Eine Instanz eines binären CSP heißt *kantenkonsistent* (oder 2-*konsistent*), wenn alle Kanten im Constraintgraphen kantenkonsistent sind.

Ein einfacher Algorithmus zur Lösung von CSPs ist Backtracking (siehe z.B. [7, Abschnitt 2.3]). Ist die CSP Instanz sowohl knoten- als auch kantenkonsistent und der Constraintgraph ein Baum, so existiert eine Backtrack-freie Suchstrategie. Daher ist man an Verfahren interessiert, die Knoten-, Kanten- und Pfadkonsistenz in einem CSP erzeugen. Knotenkonsistenz kann nahezu trivial erreicht werden. Kanten- und Pfadkonsistenz sind deutlich schwieriger.

Im Kalkül von Allen (siehe etwa [7, 8]) wird auf Intervalle als Primitive zurückgegriffen. Allen definiert die sieben Grundrelationen zwischen nichtentarteten abgeschlossenen Intervallen $X = [x_1, x_2]$ und $Y = [y_1, y_2]$, die dann durch Bildung von inversen Relationen in offensichtlicher Weise[1] auf 13 Relationen ausgedehnt werden.

Um die Grundbereiche endlich zu halten, seien für das Projekt dabei die Intervallgrenzen ganze Zahlen aus $[0, 2n]$, wobei n die Anzahl der Intervallvariablen ist. Das bedeutet, daß der Grundbereich für eine Intervallvariable X gleich

$$D_X = \{\, [a, b] : a, b \in \mathbb{N} \text{ und } 0 \leq a < b \leq 2n \,\}$$

ist.

2.3.3 Algorithmen

Algorithmus „Arc-Consistency“ (AC) und Varianten Die Algorithmen AC-1, AC-3 und AC-4 (siehe z.B. [7, Kapitel 4]) erzeugen aus einer Instanz eines CSP eine äquivalente Instanz (d.h. mit gleicher Lösungsmenge), die zugleich kantenkonsistent sind. Diese Algorithmen entfernen „redundante“ Werte aus den Grundbereichen der Variablen.

Algorithmus „Path-Consistency“ (PC) und Varianten Die Algorithmen PC-1, PC-2 und PC-4 (siehe z.B. [7, Kapitel 4]) erzwingen Pfadkonsistenz. Im Gegensatz zu den Algorithmen AC entfernen sie nicht nur redundante Werte aus den Grundbereichen, sondern auch redundante Compound Labels aus den Constraints (Dabei wird ein binäres Constraint als eine Menge von zulässigen 2-Compound Labeln repräsentiert).

Backtracking und Forward-Checking Backtracking (siehe z.B. [7, Abschnitt 2.3, Kapitel 5]) ist eine generelle Lösungsstrategie für CSPs. Die Laufzeit hängt dabei stark von der Größe der einzelnen Grundbereiche ab. Eine Variante von Backtracking ist Forward-Checking [7, Kapitel 5].

2.3.4 Aufgaben

Die oben genannten Algorithmen sollen implementiert werden. Untersucht werden soll dann, wie effizient die einzelnen Algorithmen sind, wenn man sie als Preprocessing-Verfahren für Backtracking und Forward-Checking einsetzt. Dabei soll neben der Rechenzeit für das abschließende Backtracking / Forward-Checking auch die Gesamtzeit inklusive des Preprocessings berücksichtigt werden (cf. [7, Abschnitt 2.3.4]).

Als Klasse von qualitativen CSPs sollen bei der Untersuchung Allens temporale Intervall-Constraintnetze (siehe etwa [8]) verwendet werden.

[1] Die inverse Relation r' zu r ist definiert durch $X\, r'\, Y \;:\Leftrightarrow\; Y\, r\, X$.

2.3.5 Eingabespezifikation

Allens Constraintnetze sind binäre CSP, die wieder als bewertete Graphen darzustellen sind. Eine Instanz soll als bewerteter Graph in einer ASCII-Datei in folgendem Format eingegeben werden:

Ein Constraint $X_i\{\mathrm{op}_1, \dots, \mathrm{op}_r\}X_j$ wird als Zeile

$$i \quad j : r \quad \mathrm{op}_1 \quad \mathrm{op}_2 \cdots \mathrm{op}_r$$

dargestellt.

Die Eingabedatei soll zunächst die Anzahl n der Variablen enthalten (da Allens Constraintnetze vollständige Graphen sind, ergibt sich die Anzahl der Constraints aus der Anzahl n der Variablen). Danach folgt die Spezifikation der Constraints im oben genannten Format.

2.3.6 Aussagespezifikation

Die einzelnen Algorithmen zum Erzwingen von Konsistenz sollen das von ihnen erzeugte Constraint-Netz in einer Ausgabedatei im oben geschilderten Format erzeugen. Auf diese Datei ist dann die Backtracking-Suche oder das Forward-Checking anzuwenden.

Literatur

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[2] R. Dechter, I. Meiri, and J. Pearl, *Temporal constraint networks*, Artificial Intelligence, 49 (1991), pp. 61–95.

[3] C. M. Papadimitriou, *Computational Complexity*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[4] R. Rodosek and I. Schiermeyer, *Binary decisions for solving 3-satisfiability problems*, Discrete Applied Mathematics, (1997). To appear.

[5] U. Schöning, *Logik für Informatiker*, Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, Oxford, 4 ed., 1995.

[6] E. Schwalb and R. Dechter, *Processing disjunctions in temporal constraint networks*, Artificial Intelligence, 97 (1997), pp. 29–61.

[7] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press Limited, London, UK, 1993.

[8] P. van Beek, *Reasoning about qualitative temporal information*, Artificial Intelligence, 58 (1992), pp. 297–326.