Wissensbasierte Systeme

Vorlesung 4 vom 03.11.2004

Sebastian Iwanowski
FH Wedel

Wissensbasierte Systeme

1. Motivation
2. Prinzipien und Anwendungen
3. ➔ Logische Grundlagen
4. Suchstrategien
5. Symptombasierte Diagnose
6. Modellbasierte Diagnose
   - Kandidatengenerierung
   - Konfliktgenerierung
   - Wertpropagierung
   - Gesamtarchitektur
   - Komponentenmodellierung
7. Weitere Wissensrepräsentationsformen
8. Bewertung wissensbasierter Systeme

Wdh.: Prädikatenlogik

Bsp.: $\exists\, x, y \in R: ((2 < x < 4) \wedge (0 < y < 6) \wedge (x + y > 7) \wedge (x \cdot y < 10))$

Die Prädikatenlogik (1. Stufe) erweitert die Aussagenlogik um folgende Elemente:

- **Prädikate**
  - Aussagen, die von Variablen abhängen
    (wenn es von k Variablen abhängt,
    dann heißt das Prädikat k-stellig)
- **Variable**
  - entsprechen den Literalen der Aussagenlogik,
    können aber beliebig viele Werte annehmen
- **Funktionen**
  - eindeutige Zuordnungen, die von Variablen abhängen
    (wenn sie von k Variablen abhängt,
    dann heißt die Funktion k-stellig)
  - 0-stellige Funktionen sind Konstante
- **Quantoren**
  - Existenzquantor ($\exists$) und Allquantor ($\forall$)
  - Quantoren werden nur auf Variablen angewendet (sonst nicht 1. Stufe)

Wdh.: Prädikatenlogische Formeln

- **Eine prädikatenlogische Formel ist eine Verknüpfung von endlich vielen Variablen, Funktionen und Prädikaten mit aussagenlogischen Operatoren oder Quantoren, die sich nur auf Variable beziehen.**

Bsp.: $\forall x\ (R(\mathbf{y}, \mathbf{z}) \wedge \exists y\ (\neg P(\mathbf{y}, \mathbf{x}) \vee R(\mathbf{y}, \mathbf{z})))$

Grüne Vorkommen von y und z sind **frei**.
Rote Vorkommen von x, y und z sind **gebunden**.

Geschlossene Formeln:
Formeln, die keine freien Variablen enthalten.

Offene Formeln:
Formeln, die keine gebundenen Variablen enthalten.

Substitution:
$\varphi\,[x/t]$ bezeichnet diejenige Formel, die aus φ entsteht, wenn alle freien Vorkommen von x durch Term t ersetzt werden.

Wdh.: Prädikatenlogische Formeln

- **Eine Belegung einer Formel ist eine Zuweisung von *Werten aus festgelegten Definitionsbereichen an die freien Variablen* derart, dass dieselben Variablen immer denselben Wert erhalten.**

- **Eine Formel heißt erfüllbar, wenn es eine Belegung gibt derart, dass die Formel wahr ist.**

  - Das Erfüllbarkeitsproblem ist in der Prädikatenlogik nicht entscheidbar, d.h. kein Algorithmus kann jemals in der Lage sein, von jeder Formel zu entscheiden, ob sie erfüllbar ist oder nicht.

Das allgemeine Problem ist unlösbar !

Gibt es dennoch einen Ausweg ?

Ja, löse ein spezielleres Problem !

Logisches Programmieren für die Prädikatenlogik

Idealziel:

Versuch, alle Folgerungen aus einer Menge prädikatenlogischer Formeln automatisch zu gewinnen

geht nicht !

Realistischeres Ziel:

Versuch, möglichen Widerspruch aus einer Menge prädikatenlogischer Formeln aufzudecken

Beispiel: $\{\neg P(x, f(y)),\ P(z, f(g(z)))\}$

Frage: Wann ergibt sich ein Widerspruch?

Antwort: Nur wenn die beiden Atome $P(x, f(y))$ und $P(z, f(g(z)))$ identifiziert werden.

Eine logische Programmiersprache braucht daher *Unifikation*:

Ersetzung der Variablen durch Terme, so dass beide Atome gleich werden.

Logisches Programmieren für die Prädikatenlogik

Das Prinzip der Unifikation:

$\sigma = [x_1/t_1, x_2/t_2, ..., x_n/t_n]$ sei eine ***Substitution***

Für eine Formel α ist $\sigma\alpha$ die ***Anwendung der Substitution*** σ auf α:

Beispiel:

Formel	α	$= P(f(x), y)$
Substitution	σ	$= [x/z, y/f(z)]$
Anwendung	$\sigma\alpha$	$= P(f(z), f(z))$

Definition:

Eine Substitution σ heißt ***Unifikator*** für die Formeln α_1 und α_2, wenn gilt: $\sigma\alpha 1 = \sigma\alpha 2$.

Beispiel:

Unifikation der Atome $Q(f(x), v, b)$ und $Q(f(a), g(u), y)$
durch die Substitution $\sigma = [x/a, v/g(u), y/b]$

Logisches Programmieren für die Prädikatenlogik

Das Prinzip der Unifikation:

<u>Satz (Existenz):</u>

Für je zwei Ausdrücke gibt es, bis auf Variablenumbenennung, entweder einen eindeutigen allgemeinsten Unifikator oder die beiden Ausdrücke sind nicht unifizierbar.

<u>Satz (Berechenbarkeit):</u>

Es gibt einen Algorithmus, der für zwei beliebige Ausdrücke entweder die Nichtunifizierbarkeit beweist oder den allgemeinsten Unifikator berechnet.

Übungsbeispiele für Unifikation:

$P(x)$ und $Q(y)$

$P(x, y)$ und $P(z)$

$P(x, y)$ und $P(a, f(a))$

$P(x, y)$ und $P(f(z), g(z))$

$P(x, f(y))$ und $P(z, f(g(z))$

$P(x, x)$ und $P(f(y), f(g(z))$

$P(x, f(x))$ und $P(y, y)$

$P(x, a)$ und $P(b, x)$

$P(x, f(x, x), z, f(z, z))$ und $P(f(a, a), y, f(y, y), u)$

Logisches Programmieren für die Prädikatenlogik

Das Prinzip der Programmiersprache PROLOG:

PROLOG versucht, mittels wiederholter und verschachtelter Anwendung
von **Resolution** und **Unifikation**
zu einer gegebenen Formelmenge einen Widerspruch zu finden.

Satz (Widerspruchsvollständigkeit):

Falls die Formelmenge widersprüchlich ist, kann man den Widerspruch immer finden.

Was fehlt ?

Satz (Äquivalenz von Folgerbarkeit und Widerspruchsaufdeckung):

Wer zu jeder Formelmenge jeden Widerspruch aufdecken kann, kann zu jeder Formelmenge und zu jeder neuen Formel sagen, ob die neue Formel aus der alten Formelmenge folgt.

Beweis ?

Was kann also Prolog auch noch ?

Logisches Programmieren für die Prädikatenlogik

Wie macht man aus PROLOG eine vollständige Programmiersprache ?

Durch Beschränkung der Eingabe !

PROLOG akzeptiert nur Mengen von Formeln der Form:

$p \wedge q \wedge ... \wedge r \rightarrow x$
Regeln (Hornklauseln)

Satz (Vollständigkeit der Resolution auf Hornklauseln):

Für jede Menge von Hornklauseln und eine neue Hornklausel kann Prolog nach endlicher Zeit entscheiden, ob die neue Hornklausel aus der alten Menge folgt oder nicht

Anmerkung: „Endliche Zeit“ kann „sehr lange“ heißen !

Darum wollen KI-Wissenschaftler alles mit Regeln formulieren !

Wissensbasierte Systeme

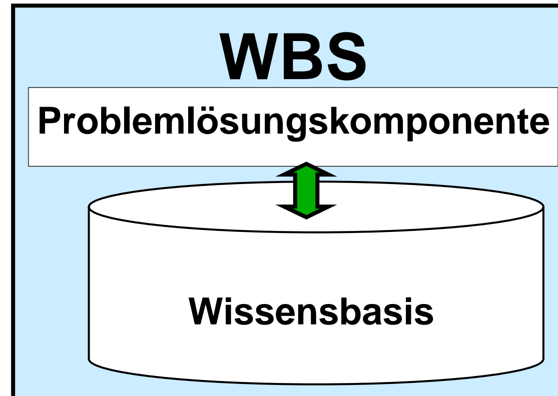
1. Motivation
2. Prinzipien und Anwendungen
3. Logische Grundlagen
4. ➔ Suchstrategien

   Symptombasierte Diagnose

5. Modellbasierte Diagnose
   - Kandidatengenerierung
   - Konfliktgenerierung
   - Wertpropagierung
   - Gesamtarchitektur
   - Komponentenmodellierung
6. Andere Diagnosemethoden
7. Weitere Wissensrepräsentationsformen
8. Bewertung wissensbasierter Systeme

Suchstrategien

Warum sind Suchstrategien so wichtig in Wissensbasierten Systemen ?



Die Problemlösungskomponente muss fast immer ein Belegungsproblem für Constraints aus der Wissensbasis lösen !

All problem solvers search

Constraint Satisfaction Problem (CSP)

Spezifikation eines CSP:

- **Variablenmenge**
- **Definitionsbereiche (Domains)**
- **Constraints: Beziehungen zwischen den Variablen**
  (in der Regel Gleichungen oder Ungleichungen)

häufig auch noch dabei:

- **weiche Constraints**
  (Constraints dürfen verletzt werden)
- **Optimierungskriterium**
  (in der Regel Funktion der Variablen, die minimiert oder maximiert werden soll)

gültige Lösung:

Belegung aller Variablen mit Werten, sodass alle harten Constraints erfüllt sind

optimale Lösung:

gültige Lösung, die das Optimierungskriterium optimiert

Constraint Satisfaction Problem (CSP)

Anwendungsbeispiele von CSP:

- Technische Diagnose
- Technische Konfiguration

Anwendungsbeispiele für andere Suchprobleme:

- Problem des Handelsreisenden (TSP)
- Problem des kürzesten Weges (Routing)
- Gewinnspiele

Suchen in Suchgraphen

Suchgraph:

- **Knoten: beschreibt Zustand in der Suchdomäne**
- **Kante: Übergang von einem Zustand in den nächsten**
  (in der Regel mit Richtung)
- **Startknoten: Anfangszustand**
  (ist immer eindeutig)
- **Zielknoten: gewünschter Endzustand (Lösung des Problems)**
  (es darf mehrere geben)

wünschenswert:

- **Suchgraph ist Suchbaum**
  (Pfad vom Startknoten zu jedem Zielknoten ist eindeutig)

Verschiedene Suchziele:

1) **Alle Lösungen eines Problems finden**
2) **Die beste Lösung finden**
3) **Ein paar gute Lösungen finden**

Lösen von CSP mit Suchbäumen

- Zustand: Belegung von Variablen mit Werten
- Folgezustand: Belegung einer weiteren Variable mit einem Wert unter Beibehaltung der Werte für die bisher belegten Variablen
- Startknoten: keine Variable hat einen Wert
- Zielknoten: gültige Lösung
- Expansion eines Knotens: Berechnen aller Folgeknoten

Verschiedene Suchstrategien unterscheiden sich in:

Welcher Knoten wird als nächstes expandiert ?

Allgemeine Suchstrategien für CSP

Im allgemeinen für CSP nur *blinde (uninformierte) Suche* möglich:

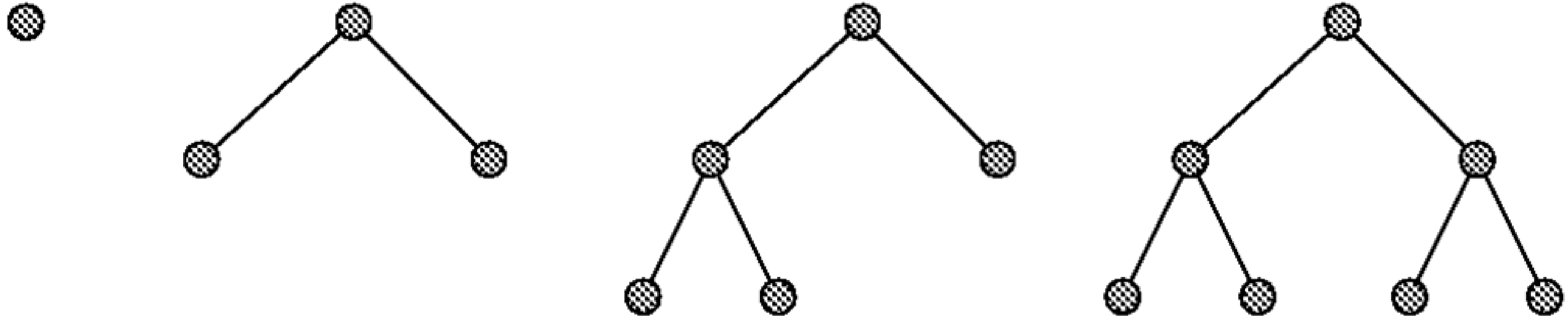
Es gibt keine Information über günstige Suchrichtungen (das Ziel wird erst bei Erreichen erkannt)

Die wichtigsten Suchstrategien:

1. Breitensuche (breadth-first-search)
2. Tiefensuche (depth-first-search)
3. Bestensuche (best-first-search)

Allgemeine Suchstrategien für CSP

Breitensuche (breadth-first-search):

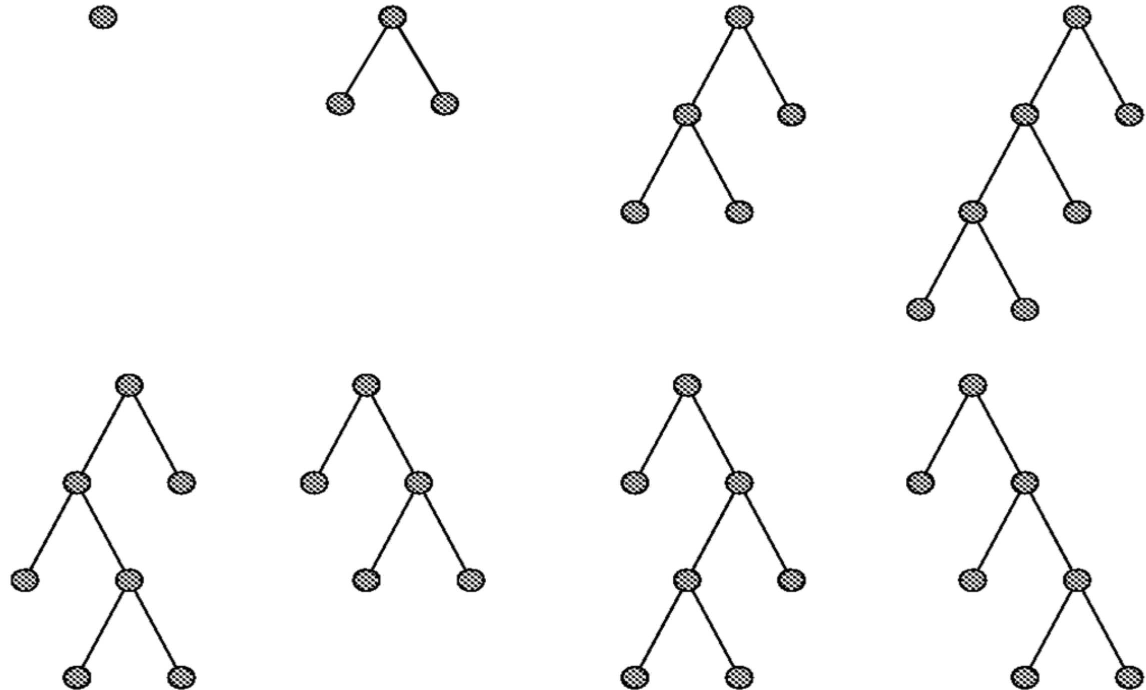


Exponentieller Aufwand für Zeit und Platz

Für CSP uninteressant

Allgemeine Suchstrategien für CSP

Tiefensuche (depth-first-search)



Exponentieller Aufwand für Zeit

Linearer Aufwand für Platz

Der „Normalfall“ für allgemeine CSP

Allgemeine Suchstrategien für CSP

Bestensuche (best-first-search)

- zusätzlich sei gegeben: Bewertungsfunktion für die Zustände
- Expandiere jeweils den Zustand mit bester Kostenbewertung

Im *schlechtesten Fall* ist das nicht besser als Tiefensuche:

Exponentieller Aufwand für Zeit

Linearer Aufwand für Platz

Der „Normalfall“ für allgemeine CSP

Bei guten Bewertungsfunktionen ist das *Durchschnittsverhalten* viel besser!

In Spezialfällen ebenfalls:

Bsp.: Spezialfall „Kürzeste-Wege-Problem“:

Algorithmus von Dijkstra (nur noch quadratischer Aufwand für Zeit)

Spezielle Suchstrategien für CSP

Zurücksetzen (Backtracking)

- Teste alle Constraints auch bei unvollständigen Variablenbelegungen
- Zustände, die irgendwelche Constraints bereits verletzen, werden nicht weiter expandiert

Vorwärtstest (Forward Checking)

- Reduziere alle Domains für alle noch nicht belegten Variablen, sodass keine Konflikte zwischen Constraints mehr entstehen.
- Setze zurück, wenn die Domains dadurch leer werden.

Beim nächsten Mal: Kandidatengenerierung bei der Modellbasierte Diagnose