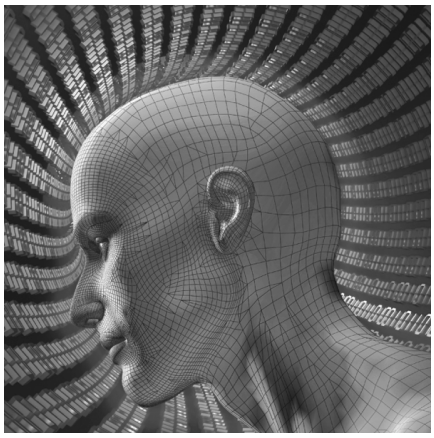
Stuart Russell
Peter Norvig

Künstliche Intelligenz

Ein moderner Ansatz

3., aktualisierte Auflage

ALWAYS LEARNING
PEARSON



Stuart Russell
Peter Norvig

Künstliche Intelligenz

Ein moderner Ansatz

3., aktualisierte Auflage

Fachliche Betreuung:
Prof. Dr. Frank Kirchner

PEARSON

Higher Education
München • Harlow • Amsterdam • Madrid • Boston
San Francisco • Don Mills • Mexico City • Sydney
a part of Pearson plc worldwide

▶ Abbildung 9.1 zeigt einen Algorithmus für die Berechnung der allgemeinsten Unifikatoren. Der Prozess ist ganz einfach: Die beiden Ausdrücke werden simultan „Seite an Seite“ rekursiv ausgewertet, wobei ein Unifikator aufgebaut wird, was aber scheitert, wenn zwei einander entsprechende Punkte in den Strukturen nicht übereinstimmen. Es gibt nur einen aufwändigen Schritt: Wenn eine Variable mit einem komplexen Term verglichen wird, muss man prüfen, ob die Variable selbst in dem Term vorkommt; ist dies der Fall, schlägt der Vergleich fehl, weil kein konsistenter Unifikator konstruiert werden kann. Zum Beispiel lässt sich $S(x)$ nicht mit $S(S(x))$ vereinheitlichen. Dieser sogenannte **Occur Check** (**Vorkommensprüfung**) bewirkt, dass die Komplexität des gesamten Algorithmus quadratisch in der Größe der zu unifizierenden Ausdrücke wird. Einige Systeme, einschließlich aller Logikprogrammiersysteme, lassen den Occur Check einfach weg und führen deshalb manchmal nicht korrekte Inferenzen aus; andere Systeme verwenden komplexere Algorithmen mit linearer Zeitkomplexität.

```
function UNIFY(x, y, θ) returns eine Substitution,
                                um x und y identisch zu machen
  inputs: x, eine Variable, Konstante, Liste oder ein Verbundausdruck
          y, eine Variable, Konstante, Liste oder ein Verbundausdruck
          θ, die bisher aufgebaute Substitution (optional, standardmäßig leer)

  if θ = Fehler then return Fehler
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return Fehler
```

```
function UNIFY-VAR(var, x, θ) returns eine Substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return Fehler
  else return füge {var/x} zu θ hinzu
```

Abbildung 9.1: Der Unifikationsalgorithmus. Der Algorithmus vergleicht elementweise die Struktur der Eingaben. Dabei wird die Substitution θ, die das Argument von UNIFY darstellt, aufgebaut und verwendet, um sicherzustellen, dass spätere Vergleiche mit früher eingerichteten Bindungen konsistent sind. In einem zusammengesetzten Ausdruck, wie etwa $F(A, B)$, ermittelt die Funktion OP das Funktionssymbol F und die Funktion ARGS ermittelt die Argumentliste (A, B).

9.2.3 Speichern und Abrufen

Den Funktionen TELL und ASK für Aufbau und Abfrage einer Wissensbasis liegen die elementareren Funktionen STORE und FETCH zugrunde. STORE(s) speichert einen Satz s in einer Wissensbasis, während FETCH(q) alle Unifikatoren zurückgibt, sodass die Abfrage q mit einem der Sätze in der Wissensbasis vereinheitlicht wird. Das Problem, das wir für die Demonstration der Unifikation verwendet haben – die Ermittlung von Fakten, die mit *Kennt*(*John*, x) unifizieren –, ist ein Beispiel für die Anwendung von FETCH.

Am einfachsten lassen sich STORE und FETCH implementieren, wenn man alle Fakten in einer langen Liste verwaltet und jede Abfrage gegen jedes Element der Liste unifiziert. Diese Vorgehensweise ist zwar ineffizient, funktioniert aber, und Sie brauchen nicht mehr, um das restliche Kapitel zu verstehen. Der restliche Abschnitt zeigt Möglichkeiten auf, das Abrufen effizienter zu machen, und kann beim ersten Durchlesen übersprungen werden.

Wir können FETCH effizienter machen, indem wir sicherstellen, dass Unifikationen nur für solche Sätze angewendet werden, für die eine *gewisse* Wahrscheinlichkeit besteht, dass sie unifiziert werden. Beispielsweise ist es sinnlos, *Kennt*(*John*, *x*) mit *Bruder*(*Richard*, *John*) zu vereinheitlichen. Derartige Unifikationen können wir vermeiden, indem wir die Fakten in der Wissensbasis **indizieren**. Ein einfaches Schema, die **Prädikatenindizierung**, legt alle *Kennt*-Fakten in einem Behälter ab, alle *Bruder*-Fakten in einem anderen. Die Behälter können in einer Hash-Tabelle abgelegt werden, um einen effizienten Zugriff zu ermöglichen.

Die Prädikatenindizierung ist sinnvoll, wenn es mehrere Prädikatssymbole, aber nur wenige Klauseln für jedes Symbol gibt. In einigen Anwendungen gibt es jedoch viele Klauseln für ein bestimmtes Prädikatssymbol. Angenommen, die Steuerbehörden wollen überwachen, wer wen beschäftigt, und verwenden dazu das Prädikat *Beschäftigt*(*x*, *y*). Das wäre ein sehr großer Behälter mit wahrscheinlich Millionen von Arbeitgebern und noch mehr Arbeitnehmern. Die Beantwortung einer Abfrage wie *Beschäftigt*(*x*, *Richard*) mithilfe der Prädikatenindizierung bedingt das Durchsuchen des gesamten Behälters.

Für diese Abfrage wäre es hilfreich, wenn die Fakten sowohl nach dem Prädikat als auch nach dem zweiten Argument indiziert wären, vielleicht unter Verwendung eines kombinierten Hashtabellen-Schlüssels. Dann könnten wir den Schlüssel einfach aus der Abfrage konstruieren und genau die Fakten finden, die mit der Abfrage unifizieren. Für andere Abfragen, wie etwa *Beschäftigt*(*IBM*, *y*), müssten wir die Fakten indizieren, indem wir das Prädikat mit dem ersten Argument kombinieren. Dazu können Fakten unter mehreren Indexschlüsseln gespeichert werden, sodass unterschiedliche Abfragen, mit denen sie vereinheitlicht werden könnten, unmittelbar zugänglich sind.

Für einen Satz, der gespeichert werden soll, ist es möglich, Indizes für *alle möglichen* Abfragen zu konstruieren, die sich mit ihm vereinigen. Für den Fakt *Beschäftigt*(*IBM*, *Richard*) lauten die Abfragen:

Beschäftigt(*IBM*, *Richard*)	Beschäftigt IBM Richard?
Beschäftigt(*x*, *Richard*)	Wer beschäftigt Richard?
Beschäftigt(*IBM*, *y*)	Wen beschäftigt IBM?
Beschäftigt(*x*, *y*)	Wer beschäftigt wen?

Diese Abfragen bilden einen **Subsumtionsverband**, wie in ▶ Abbildung 9.2(a) gezeigt. Der Verband hat einige interessante Eigenschaften. Beispielsweise wird das Kind jedes Knotens im Verband mithilfe einer einzigen Subsumtion aus seinen Eltern ermittelt; der „höchste“ gemeinsame Nachfahre von zwei Knoten ist das Ergebnis der Anwendung ihres allgemeinsten Unifikators. Der Teil des Verbandes über einem beliebigen Grundfakt kann systematisch konstruiert werden (Übung 9.5). Ein Satz mit wiederholten Konstanten hat einen etwas anderen Verband, wie in ▶ Abbildung 9.2(b) gezeigt. Funktionssymbole und Variablen in den zu speichernden Sätzen führen zu noch interessanteren Verbandsstrukturen.

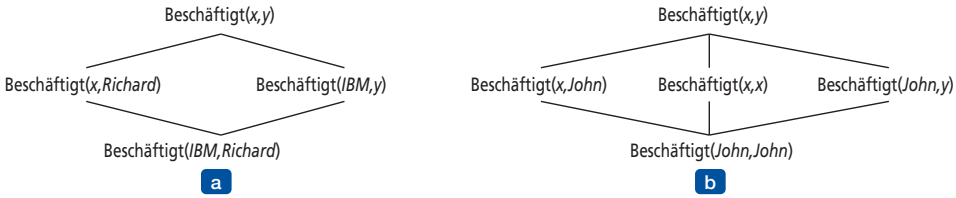


Abbildung 9.2: (a) Der Subsumtionsverband, dessen niedrigster Knoten der Satz *Beschäftigt*(*IBM*, *Richard*) ist. (b) Der Subsumtionsverband für den Satz *Beschäftigt*(*John*, *John*).

Das bisher beschriebene Schema funktioniert gut, wenn der Verband eine kleine Anzahl an Knoten enthält. Für ein Prädikat mit n Argumenten enthält der Verband $O(2^n)$ Knoten. Wenn Funktionssymbole erlaubt sind, ist die Anzahl der Knoten außerdem exponentiell in der Größe der Terme in dem zu speichernden Satz. Das kann zu einer riesigen Anzahl an Indizes führen. An einem bestimmten Punkt werden die Vorteile der Indizierung durch die Kosten für das Speichern und die Verwaltung all dieser Indizes aufgehoben. Wir können reagieren, indem wir eine feststehende Strategie anwenden, beispielsweise nur solche Schlüssel verwalten, die aus einem Prädikat plus allen Argumenten bestehen, oder eine adaptive Strategie verwenden, die Indizes erzeugt, um den Anforderungen der gestellten Abfragearten gerecht zu werden. Für die meisten KI-Systeme ist die Anzahl der zu speichernden Fakten klein genug, dass eine effiziente Indizierung als gelöstes Problem betrachtet werden kann. Für kommerzielle Datenbanken, wo die Anzahl der Fakten in die Milliarden geht, ist das Problem Gegenstand intensiver Untersuchungen und technologischer Entwicklungen gewesen.

9.3 Vorwärtsverkettung

In *Abschnitt 7.5* wurde ein Algorithmus für die Vorwärtsverkettung für aussagenlogische definite Klauseln vorgestellt. Die Idee dabei ist einfach: Man beginnt mit den atomaren Sätzen in der Wissensbasis, wendet den Modus ponens in Vorwärtsrichtung an und fügt neue atomare Sätze ein, bis keine weiteren Inferenzen gemacht werden können. Hier erklären wir, wie der Algorithmus auf definite Klauseln erster Stufe angewendet wird und wie er effizient implementiert werden kann. Definite Klauseln wie etwa *Situation* $\Rightarrow$ *Antwort* sind besonders praktisch für Systeme, die Inferenzen machen, um auf neu eingetroffene Information zu reagieren. Viele Systeme können auf diese Weise definiert werden und das Schließen mit Vorwärtsverkettung lässt sich sehr effizient implementieren.

9.3.1 Definite Klauseln erster Stufe

Die definiten Klauseln erster Stufe sind den aussagenlogischen definiten Klauseln sehr ähnlich (*Abschnitt 7.5.3*): Es handelt sich dabei um Disjunktionen von Literalen, *von denen genau eines positiv ist*. Eine definite Klausel ist entweder atomar oder sie ist eine Implikation, deren Antezedenz eine Konjunktion positiver Literale und deren Konsequenz ein einziges positives Literal ist. Nachfolgend sehen Sie definite Klauseln erster Stufe:

$$König(x) \wedge Gierig(x) \Rightarrow Böse(x)$$
$$König(John)$$
$$Gierig(y).$$

Anders als aussagenlogische Literale können Literale in der Logik erster Stufe Variablen enthalten, wobei man annimmt, dass diese Variablen allquantifiziert sind. (Normalerweise lassen wir Allquantoren weg, wenn wir definite Klauseln schreiben.) Nicht jede Wissensbasis kann in eine Menge definiter Klauseln umgewandelt werden, weil es die Einschränkung mit dem einzelnen positiven Literal gibt – aber viele können. Betrachten Sie das folgende Problem:

> Laut Gesetz ist es für einen Amerikaner ein Verbrechen, Waffen an feindliche Nationen zu verkaufen. Das Land Nono, ein Feind von Amerika, besitzt einige Raketen und alle seine Raketen wurden ihm von Colonel West verkauft, der Amerikaner ist.

Wir werden beweisen, dass West ein Krimineller ist. Zuerst werden wir diese Fakten als definite Klauseln erster Stufe repräsentieren. Der nächste Abschnitt zeigt, wie der Algorithmus für die Vorwärtsverkettung das Problem löst.

„... ist es für einen Amerikaner ein Verbrechen, Waffen an feindliche Nationen zu verkaufen“:

$$\mathit{Amerikaner}(x) \wedge \mathit{Waffe}(y) \wedge \mathit{Verkauft}(x, y, z) \wedge \mathit{Feindlich}(z) \Rightarrow \mathit{Kriminell}(x). \quad (9.3)$$

„Nono ... besitzt einige Raketen.“ Der Satz $\exists x\, \mathit{Besitzt}(\mathit{Nono}, x) \wedge \mathit{Rakete}(x)$ wird durch die Existentielle Instantiierung in zwei definite Klauseln umgewandelt, wobei eine neue Konstante M_1 eingeführt wird:

$$\mathit{Besitzt}(\mathit{Nono}, M_1) \quad (9.4)$$
$$\mathit{Rakete}(M_1) \quad (9.5)$$

„Alle seine Raketen wurden ihm von Colonel West verkauft“:

$$\mathit{Rakete}(x) \wedge \mathit{Besitzt}(\mathit{Nono}, x) \Rightarrow \mathit{Verkauft}(\mathit{West}, x, \mathit{Nono}). \quad (9.6)$$

Außerdem müssen wir wissen, dass Raketen Waffen sind:

$$\mathit{Rakete}(x) \Rightarrow \mathit{Waffe}(x). \quad (9.7)$$

Und wir müssen wissen, dass ein Feind Amerikas als „feindlich“ gilt:

$$\mathit{Feind}(x, \mathit{Amerika}) \Rightarrow \mathit{Feindlich}(x). \quad (9.8)$$

„West, der Amerikaner ist ...“:

$$\mathit{Amerikaner}(\mathit{West}). \quad (9.9)$$

„Das Land Nono, ein Feind Amerikas ...“:

$$\mathit{Feind}(\mathit{Nono}, \mathit{Amerika}). \quad (9.10)$$

Diese Wissensbasis enthält keine Funktionssymbole und ist deshalb eine Instanz der Klasse der **Datalog**-Datenbanken. Datalog ist eine Sprache, die auf definite Klauseln erster Stufe ohne Funktionssymbole beschränkt ist. Ihr Name ergibt sich daraus, dass sie die Art der Anweisungen darstellen kann, die in relationalen Datenbanken üblich sind. Wir werden sehen, dass das Fehlen von Funktionssymbolen die Inferenz sehr viel einfacher macht.

9.3.2 Ein einfacher Algorithmus für die Vorwärtsverkettung

Der erste Algorithmus für die Vorwärtsverkettung, den wir hier betrachten wollen, ist sehr einfach, wie in ▶ Abbildung 9.3 gezeigt. Beginnend bei den bekannten Fakten führt er alle Regeln aus, deren Prämissen erfüllt sind, und fügt ihre Schlüsse den bekannten Fakten hinzu. Der Prozess wird wiederholt, bis die Abfrage beantwortet ist (vorausgesetzt, es ist nur eine Antwort erforderlich) oder bis keine neuen Fakten mehr hinzugefügt werden. Beachten Sie, dass ein Fakt nicht „neu" ist, wenn er nur eine **Umbenennung** eines bekannten Fakts darstellt. Ein Satz ist eine Umbenennung eines anderen, wenn die Sätze bis auf die Namen der Variablen identisch sind. Beispielsweise sind *Mag*(*x*, *Eis*) und *Mag*(*y*, *Eis*) Umbenennungen voneinander, weil sie sich nur in der Wahl von *x* oder *y* unterscheiden; ihre Bedeutungen sind identisch: Jeder mag Eis.

```
function FOL-FC-Ask(KB, α) returns eine Substitution oder false
  inputs: KB, die Wissensbasis, eine Menge definiter Klauseln erster Stufe
          α, die Abfrage, ein atomarer Satz
  local variables: new, die neuen Sätze, die bei jeder Iteration abgeleitet
                   werden

  repeat until new ist leer
    new ← { }
    for each rule in KB do
      (p1 ∧ … ∧ pn ⇒ q) ← Standardize-Variables(rule)
      for each θ derart, dass Subst(θ, p1 ∧ … ∧ pn) = Subst(θ, p1' ∧ … ∧ pn')
                  für einige p1', …, pn' in KB
        q' ← Subst(θ, q)
        if q' wird nicht unifiziert mit einem in KB enthaltenen Satz oder new then
          füge q' zu new hinzu
          ϕ ← Unify(q', α)
          if ϕ ist nicht fail then return ϕ
    füge new zu KB hinzu
  return false
```

Abbildung 9.3: Ein konzeptionell einfacher, aber sehr ineffizienter Algorithmus für die Vorwärtsverkettung. Bei jeder Iteration fügt er zu *KB* die atomaren Sätze hinzu, die in einem Schritt von den Implikationssätzen und den bereits in *KB* enthaltenen atomaren Sätzen abgeleitet werden können. Die Funktion Standardize-Variables ersetzt alle Variablen in ihren Argumenten durch die neuen, die vorher noch nicht verwendet wurden.

Wir wenden wieder unser Problem aus der Welt der Verbrecher an, um zu zeigen, wie FOL-FC-Ask funktioniert. Die Implikationssätze sind (9.3), (9.6), (9.7) und (9.8). Es sind zwei Iterationen erforderlich:

- In der ersten Iteration hat Regel (9.3) nicht erfüllte Prämissen.

  Regel (9.6) ist erfüllt mit $\{x / M_1\}$ und *Verkauft*(*West*, M_1, *Nono*) wird hinzugefügt.

  Regel (9.7) ist erfüllt mit $\{x / M_1\}$ und *Waffe*(M_1) wird hinzugefügt.

  Regel (9.8) ist erfüllt mit {*x* / *Nono*} und *Feindlich*(*Nono*) wird hinzugefügt.

- In der zweiten Iteration ist Regel (9.3) mit $\{x / West, y / M_1, z / Nono\}$ erfüllt und *Kriminell*(*West*) wird hinzugefügt.

▶ Abbildung 9.4 zeigt den erzeugten Beweisbaum. Beachten Sie, dass an dieser Stelle keine neuen Inferenzen mehr möglich sind, weil jeder Satz, der durch die Vorwärtsverkettung geschlossen werden konnte, bereits explizit in der Wissensbasis enthalten ist. Eine solche Wissensbasis wird auch als **Fixpunkt** des Inferenzprozesses bezeichnet. Fix-

punkte, die durch Vorwärtsverkettung mit definiten Klauseln erster Stufe erzielt werden, sind denen für die aussagenlogische Vorwärtsverkettung (*Abschnitt 7.5.4*) sehr ähnlich; der wichtigste Unterschied ist, dass ein Fixpunkt erster Stufe allquantifizierte atomare Sätze enthalten kann.

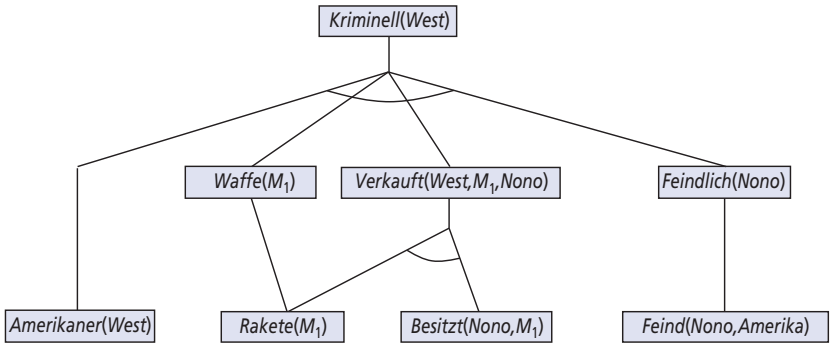


Abbildung 9.4: Der Beweisbaum, der durch die Vorwärtsverkettung für das Verbrecherbeispiel erzeugt wird. Die anfänglichen Fakten erscheinen auf der untersten Ebene, Fakten, die während der ersten Iteration abgeleitet wurden, in der mittleren Ebene und Fakten, die während der zweiten Iteration abgeleitet wurden, auf der obersten Ebene.

FOL-FC-Ask ist einfach zu analysieren. Erstens ist die Funktion **korrekt**, weil jede Inferenz einfach nur eine Anwendung des Verallgemeinerten Modus ponens ist, der korrekt ist. Zweitens ist sie **vollständig** für Wissensbasen mit definiten Klauseln; das bedeutet, sie beantwortet jede Abfrage, deren Antworten logische Konsequenzen einer Wissensbasis mit definiten Klauseln sind. Für Datalog-Wissensbasen, die keine Funktionssymbole enthalten, ist der Beweis der Vollständigkeit relativ einfach. Wir beginnen damit, die Anzahl möglicher Fakten zu zählen, die hinzugefügt werden können, was die Anzahl der Iterationen bestimmt. Es sei k die maximale **Stelligkeit** (Anzahl der Argumente) eines beliebigen Prädikats, p die Anzahl der Prädikate und n die Anzahl der Konstantensymbole. Offensichtlich kann es nicht mehr als pn^k unterschiedliche Grundfakten geben; der Algorithmus muss also nach so vielen Iterationen einen Fixpunkt erreichen. Dann können wir ein Argument erzeugen, das ganz ähnlich dem Beweis der Vollständigkeit für die aussagenlogische Vorwärtsverkettung ist (*Abschnitt 7.5.4*). Die Details, wie der Übergang von der aussagenlogischen Vollständigkeit zur Vollständigkeit in der Logik erster Stufe erfolgen kann, sind für den Resolutionsalgorithmus in *Abschnitt 9.5* beschrieben.

Für allgemeine definite Klauseln mit Funktionssymbolen kann FOL-FC-Ask unendlich viele neue Fakten erzeugen; deshalb müssen wir hier sorgfältiger sein. Für den Fall, in dem eine Antwort auf den Abfragesatz q eine logische Konsequenz der Wissensbasis ist, müssen wir den Satz von Herbrand bemühen, um zu behaupten, dass der Algorithmus einen Beweis findet. (Weitere Informationen für den Resolutionsfall finden Sie in *Abschnitt 9.5*.) Gibt es für die Abfrage keine Antwort, kann es sein, dass der Algorithmus in einigen Fällen nicht terminiert. Beinhaltet die Wissensbasis beispielsweise die Peano-Axiome

$$NatNum(0)$$
$$\forall n \quad NatNum(n) \Rightarrow NatNum(S(n))$$

fügt die Vorwärtsverkettung $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$ usw. ein. Dieses Problem ist im Allgemeinen nicht zu vermeiden. Wie bei der allgemeinen Logik erster Stufe ist die logische Konsequenz bei definiten Klauseln semientscheidbar.

9.3.3 Effiziente Vorwärtsverkettung

Der in Abbildung 9.3 gezeigte Algorithmus für die Vorwärtsverkettung soll Ihnen helfen, das Konzept zu verstehen, und ist nicht auf die Effizienz der Operation ausgelegt. Es gibt drei mögliche Ursachen für die Komplexität. Erstens, die „innere Schleife“ des Algorithmus sucht alle möglichen Unifikatoren, sodass die Prämisse einer Regel mit einer geeigneten Menge von Fakten in der Wissensbasis unifiziert. Man spricht häufig von einem **Mustervergleich** (**Pattern Matching**), was sehr aufwändig sein kann. Zweitens, der Algorithmus überprüft jede Regel bei jeder Iteration erneut, um festzustellen, ob ihre Prämissen erfüllt sind, selbst wenn der Wissensbasis bei jeder Iteration nur sehr wenige Einträge hinzugefügt werden. Und drittens kann der Algorithmus viele Fakten erzeugen, die für das Ziel nicht relevant sind. Wir werden jede dieser Ursachen nacheinander betrachten.

Regeln mit bekannten Fakten vergleichen

Das Problem, die Prämisse einer Regel mit den Fakten in der Wissensbasis zu vergleichen, scheint recht einfach zu sein. Angenommen, wir wollen die folgende Regel anwenden:

$$Rakete(x) \Rightarrow Waffe(x).$$

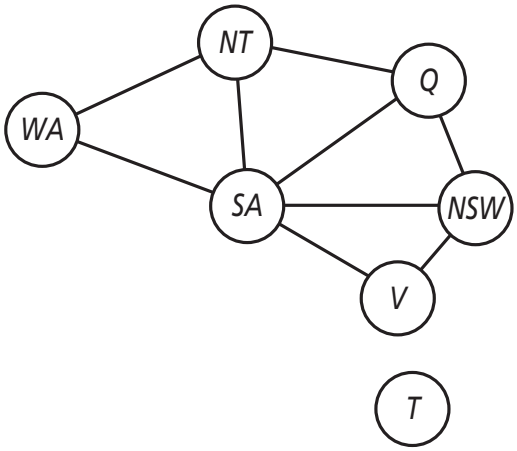
Wir müssen alle Fakten finden, die mit $Rakete(x)$ unifizieren; in einer sinnvoll indizierten Wissensbasis kann dies in konstanter Zeit pro Fakt erfolgen. Betrachten Sie jetzt die folgende Regel:

$$Rakete(x) \wedge Besitzt(Nono, x) \Rightarrow Verkauft(West, x, Nono).$$

Wieder können wir alle Objekte, die Nono besitzt, in konstanter Zeit pro Objekt finden; anschließend könnten wir für jedes Objekt überprüfen, ob es sich um eine Rakete handelt. Wenn die Wissensbasis viele Objekte enthält, die Nono besitzt, aber sehr wenige Raketen, wäre es jedoch besser, zuerst alle Raketen zu suchen und dann zu überprüfen, ob sie Nono gehören. Dies ist das Problem der **Reihenfolge von Konjunkten**: eine Reihenfolge zu finden, um die Konjunkte der Regelprämisse zu lösen, sodass die Gesamtkosten minimiert werden. Es stellt sich heraus, dass die Ermittlung einer optimalen Reihenfolge NP-hart ist, aber es gibt gute Heuristiken. Beispielsweise würde die **MRV-Heuristik** (Minimum Remaining Values, Minimum an verbleibenden Werten), die in *Kapitel 6* für CSPs verwendet wurde, vorschlagen, die Konjunkte so zu sortieren, dass zuerst nach Raketen gesucht wird, wenn es weniger Raketen als von Nono besessene Objekte gibt.

Tipp

Die Verbindung zwischen Pattern Matching und CSP ist sehr eng. Wir können jedes Konjunkt als Beschränkung der darin enthaltenen Variablen betrachten – z.B. ist $Rakete(x)$ eine unäre Beschränkung für x. Wenn wir diese Idee fortführen, *können wir jedes CSP mit endlicher Domäne als einzelne definite Klausel zusammen mit einigen ihr zugeordneten Grundfakten ausdrücken*. Betrachten Sie das Problem der Karteneinfärbung aus Abbildung 6.1, das in ▶ Abbildung 9.5(a) noch einmal gezeigt ist. ▶ Abbildung 9.5(b) zeigt eine äquivalente Formulierung als einzelne definite Klausel. Offensichtlich kann der Schluss *Colorable*() nur dann abgeleitet werden, wenn das CSP eine Lösung hat. Weil CSPs im Allgemeinen 3-SAT-Probleme als Sonderfälle beinhalten, können wir schließen, *dass der Vergleich einer definiten Klausel mit einer Menge von Fakten NP-hart ist*.



a

$$Diff(wa, nt) \wedge Diff(wa, sa) \wedge Diff(nt, q) \wedge Diff(nt, sa) \wedge Diff(q, nsw) \wedge Diff(q, sa) \wedge Diff(nsw, v) \wedge Diff(nsw, sa) \wedge Diff(v, sa) \Rightarrow Colorable()$$

$$Diff(Rot, Blau) Diff(Rot, Grün)$$
$$Diff(Grün, Rot) Diff(Grün, Blau)$$
$$Diff(Blau, Rot) Diff(Blau, Grün)$$

b

Abbildung 9.5: (a) Constraint-Graph für die Einfärbung der Landkarte von Australien. (b) Das CSP für das Karteneinfärbungsproblem, ausgedrückt als einzelne definite Klausel. Jeder Kartenbereich wird als Variable dargestellt, deren Wert eine der Konstanten *Rot*, *Grün* oder *Blau* sein kann.

Es mag etwas deprimierend erscheinen, dass die Vorwärtsverkettung ein NP-hartes Vergleichsproblem in seiner inneren Schleife aufweist. Es gibt jedoch drei Möglichkeiten, die uns aufmuntern könnten:

- Wir können uns daran erinnern, dass die meisten Regeln in echten Wissensbasen klein und einfach sind (wie die Regeln in unserem Verbrecherbeispiel) und nicht groß und kompliziert (wie die CSP-Formulierung in Abbildung 9.5). In der Datenbankwelt wird häufig vorausgesetzt, dass sowohl die Größe der Regeln als auch die Stelligkeit der Prädikate durch eine Konstante begrenzt ist und dass man sich nur über die **Datenkomplexität** Gedanken machen muss – d.h. die Komplexität der Inferenz als Funktion der Anzahl der Grundfakten in der Datenbank.
- Wir können Subklassen von Regeln einrichten, für die der Vergleich effizient ist. Im Wesentlichen kann jede Datalog-Klausel als Definition eines CSP betrachtet werden; der Vergleich ist also dann behandelbar, wenn das entsprechende CSP behandelbar ist. *Kapitel 6* beschreibt mehrere behandelbare Familien von CPSs. Wenn zum Beispiel der Constraint-Graph (der Graph, dessen Knoten Variablen sind und dessen Kanten die Beschränkungen darstellen) einen Baum bildet, kann das CSP in linearer Zeit gelöst werden. Genau das Gleiche gilt für den Regelvergleich. Wenn wir beispielsweise Südaustralien aus der Karte in Abbildung 9.5 entfernen, erhalten wir die resultierende Klausel

  $$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable().$$

  Das entspricht dem in Abbildung 6.12 gezeigten reduzierten CSP. Algorithmen für die Lösung baumstrukturierter CSPs können direkt auf das Problem des Regelvergleiches angewendet werden.
- Wir können probieren, redundante Regelvergleichsversuche im Algorithmus für die Vorwärtsverkettung zu eliminieren, was Thema des nächsten Abschnittes sein wird.

Inkrementelle Vorwärtsverkettung

Als wir die Vorwärtsverkettung anhand des Verbrecherbeispiels gezeigt haben, haben wir geschummelt: Insbesondere ließen wir Regelvergleiche weg, die der in Abbildung 9.3 gezeigte Algorithmus ausführt. In der zweiten Iteration etwa vergleicht die Regel

$$Rakete(x) \Rightarrow Waffe(x)$$

Tipp

(erneut) mit $Rakete(M_1)$ und natürlich ist der Schluss $Waffe(M_1)$ bereits bekannt; deshalb passiert nichts. Solche redundanten Regelvergleiche können vermieden werden, wenn wir die folgende Beobachtung berücksichtigen: *Jeder neue Fakt, der während der Iteration t abgeleitet wird, muss von mindestens einem neuen Fakt abgeleitet sein, der während der Iteration* $t - 1$ *abgeleitet wurde*. Das trifft zu, weil jede Inferenz, die keinen neuen Fakt aus Iteration $t - 1$ zieht, dies bereits in Iteration $t - 1$ getan haben könnte.

Diese Beobachtung führt zu einem Algorithmus für eine inkrementelle Vorwärtsverkettung, wobei wir während der Iteration t eine Regel nur dann überprüfen, wenn ihre Prämisse ein Konjunkt p_i enthält, das mit einem Fakt p_i' unifiziert, der während der Iteration $t - 1$ neu abgeleitet wurde. Der Schritt des Regelvergleiches verändert dann p_i, sodass es mit p_i' übereinstimmt, erlaubt es aber, dass die anderen Konjunkte der Regel mit Fakten aus beliebigen vorhergehenden Iterationen übereinstimmen. Dieser Algorithmus erzeugt genau dieselben Fakten bei jeder Iteration wie der in Abbildung 9.3 gezeigte, ist aber effizienter.

Mit einer geeigneten Indizierung ist es ganz einfach, alle Regeln zu identifizieren, die von jedem beliebigen Fakt ausgelöst werden können, und viele reale Systeme arbeiten tatsächlich in einem „Aktualisierungsmodus“, wo bei jedem neuen (mit TELL veranlassten) Eintrag eines Fakts im System eine Vorwärtsverkettung stattfindet. Die Inferenz durchläuft kaskadenförmig die Regelmenge, bis der Fixpunkt erreicht ist. Anschließend verarbeitet der Prozess den nächsten neuen Fakt.

Normalerweise wird nur ein kleiner Teil der Regeln in der Wissensbasis ausgelöst, wenn ein bestimmter Fakt hinzugefügt wird. Das bedeutet, es erfolgt viel redundante Arbeit beim wiederholten Aufbau partieller Vergleiche, die nicht erfüllte Prämissen aufweisen. Unser Verbrecherbeispiel ist zu klein, um dies zu verdeutlichen, aber beachten Sie, dass während der ersten Iteration eine partielle Übereinstimmung zwischen der Regel

$$Amerikaner(x) \wedge Waffe(y) \wedge Verkauft(x, y, z) \wedge Feindlich(z) \Rightarrow Kriminell(x)$$

und dem Fakt $Amerikaner(West)$ konstruiert wird. Diese partielle Übereinstimmung wird dann verworfen und in der zweiten Iteration (wenn die Regel erfolgreich war) wieder aufgebaut. Es wäre besser, die partiellen Übereinstimmungen aufzubewahren und schrittweise zu vervollständigen, wenn neue Fakten eintreffen, anstatt sie immer zu verwerfen.

Der **Rete**-Algorithmus[3] war der erste Algorithmus, der ernsthaft versuchte, dieses Problem zu lösen. Der Algorithmus leistet eine Vorverarbeitung der Regelmenge in der Wissensbasis, um eine Art Datenflussnetz zu konstruieren, in dem jeder Knoten ein Literal aus einer Regelprämisse ist. Variablenbindungen durchfließen das Netz und werden ausgefiltert, wenn sie nicht mit einem Literal übereinstimmen. Verwenden zwei Literale in einer Regel dieselbe Variable – z.B. $Verkauft(x, y, z) \wedge Feindlich(z)$ im Verbrecherbeispiel –, werden die Bindungen aus jedem Literal durch einen Gleichheitsknoten gefiltert.

3 Rete ist das lateinische Wort für Netz.

Eine Variablenbindung, die einen Knoten für ein n-stelliges Literal erreicht, wie etwa *Verkauft*(x, y, z), muss möglicherweise warten, bis Bindungen für die anderen Variablen eingerichtet sind, bevor der Prozess fortgesetzt werden kann. Der Zustand eines solchen Rete-Netzes berücksichtigt zu jedem Zeitpunkt alle partiellen Übereinstimmungen der Regeln und vermeidet damit viele wiederholte Berechnungen.

Rete-Netzwerke und verschiedene Verbesserungen daran waren eine der Schlüsselkomponenten der sogenannten **Produktionssysteme**, die zu den frühesten Vorwärtsverkettungssystemen gehörten, die ganz allgemein eingesetzt wurden.[4] Das XCON-System (ursprünglich als R1 bezeichnet, McDermott, *1982*) wurde unter Verwendung einer Produktionssystemarchitektur erstellt. XCON enthielt mehrere Tausend Regeln für den Entwurf verschiedener Konfigurationen aus Computerkomponenten der Digital Equipment Corporation. Es war einer der ersten klaren kommerziellen Erfolge auf dem sich entwickelnden Gebiet der Expertensysteme. Viele andere, ähnliche Systeme wurden unter Verwendung derselben zugrunde liegenden Technologie entwickelt, die in der allgemeinen Sprache OPS-5 implementiert wurde.

Produktionssysteme sind auch bekannt in **kognitiven Architekturen** – d.h. Modellen menschlichen Schließens –, wie z.B. ACT (Anderson, 1983) oder SOAR (Laird et al., 1987). In solchen Systemen bildet der „Arbeitsspeicher" des Modells das Kurzzeitgedächtnis des Menschen nach und die Produktionen sind Teil des Langzeitgedächtnisses. Bei jedem Operationszyklus werden Produktionen mit dem Arbeitsspeicher verglichen, in dem die Fakten enthalten sind. Eine Produktion, deren Bedingungen erfüllt sind, kann Fakten im Arbeitsspeicher hinzufügen oder daraus entfernen. Im Gegensatz zur typischen Situation in Datenbanken haben Produktionssysteme häufig viele Regeln und relativ wenige Fakten. Mit einer geeignet optimierten Vergleichstechnologie können einige moderne Systeme mit über mehreren zehn Millionen Regeln in Echtzeit arbeiten.

Irrelevante Fakten

Die letzte Ursache für Ineffizienzen bei der Vorwärtsverkettung scheint charakteristisch für den Ansatz zu sein und tritt auch im aussagenlogischen Kontext auf. Die Vorwärtsverkettung trifft alle erlaubten Inferenzen basierend auf den bekannten Fakten, *selbst wenn diese irrelevant für das vorliegende Ziel sind*. In unserem Verbrecherbeispiel gab es keine Regeln, die irrelevante Schlüsse ziehen konnten, das Fehlen der Richtungsorientierung war also kein Problem. In anderen Fällen (d.h. wenn viele Regeln die Essgewohnheiten von Amerikanern und die Preise von Raketen beschreiben) erzeugt FOL-FC-ASK viele irrelevante Schlüsse.

Eine Möglichkeit, irrelevante Schlüsse zu vermeiden, ist die Verwendung der Rückwärtsverkettung, die in *Abschnitt 9.4* beschrieben ist. Eine weitere Lösung wäre, die Vorwärtsverkettung auf eine ausgewählte Untermenge an Regeln zu begrenzen, wie es in PL-FC-ENTAILS? (*Abschnitt 7.5.4*) der Fall ist. Ein dritter Ansatz hat sich im Bereich der **deduktiven Datenbanken** herausgebildet. Hier handelt es sich um sehr große Datenbanken, ähnlich relationalen Datenbanken, aber mit Verwendung der Vorwärtsverkettung als Standardwerkzeug für Inferenz anstelle von SQL-Abfragen. Die Idee dabei ist, die Regelmenge mithilfe der Informationen vom Ziel neu zu schreiben, sodass nur relevante Variablenbindungen – diejenigen, die zu einer sogenannten **magischen Menge**

4 Das Wort **Produktion** in **Produktionssystem** bezeichnet eine Bedingung/Aktion-Regel.

Copyright