BotIshelly

Intelligente Internet-Softbots

Endbericht der Projektgruppe 343

1999/2000

BotIshelly

Intelligente Internet–Softbots

Endbericht der Projektgruppe 343

Sommersemester 1999
Wintersemester 1999/2000

Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Künstliche Intelligenz
44 221 Dortmund

http://www-ai.cs.uni-dortmund.de

Teilnehmer der Projektgruppe:
Michael Banken
Christian Fischbach
Oliver Geppert
Markus Hövener
Jens Jägersküpper
Volkher Kaschlun
Nils Malzahn
André Masloch
Ursula Mentel
Marina Podvoiskaia
Niels Schröter

Betreuer:
Dipl.–Inform. Thorsten Joachims
Dipl.–Inform. Ralf Klinkenberg

Inhaltsverzeichnis

Vorwort: Was und Wie

Dies ist der Endbericht der Projektgruppe 343, die sich ein Jahr lang mit der Erstellung einer Bibliothek zur Unterstützung der Entwicklung von Agenten für die Suche im Internet beschäftigt hat. Dieser Bericht ist in die drei folgenden Teile unterteilt:

PG 343 — Intelligente Internet–Softbots
: In diesem Abschnitt werden die einzelnen Phasen, die die Projektgruppe (PG) durchlaufen hat, beschrieben. Mit Ausnahme der Einleitung geschieht dies in chronologischer Reihenfolge. Die jeweiligen Texte spiegeln also den Stand der Dinge in der jeweiligen Phase wider. So kann es durchaus passieren, daß sich Ansichten — genau wie im zeitlichen Ablauf der PG — im Verlauf des Berichtes ändern. Dies ist durchaus so gewollt, um den Entwicklungsprozeß, den die PG durchlaufen hat, zu verdeutlichen.

Benutzungshandbuch BotIshelly
: Dieser Teil ist das Handbuch der Bibliothek, die während der PG entstanden ist. Zu der "Bedienungsanleitung" unseres "Produktes" gehört außerdem die von *JavaDoc* erzeugte HTML–Dokumentation der Java–Klassen unserer Bibliothek.

Benutzungshandbuch Firmenagent
: Um die Funktionalität unserer Bibliothek zu demonstrieren, hat die PG sie dazu benutzt, um einen Agenten zu implementieren, der die Homepages von Firmen im Internet finden soll. Dieser Teil des Berichts ist das Handbuch zu diesem Agenten, in dem dessen Benutzung und einige implementierungstechnische Details beschrieben werden.

Teil I

PG 343 — Intelligente Internet–Softbots

Inhaltsangabe Teil I

Kapitel 1

Projektbeschreibung und Themenaufbereitung

1.1 Einführung und Motivation

Elektronische Medien und speziell das World Wide Web (WWW) gewinnen an immer stärkerer Bedeutung sowohl für gewerbliche Anwendungen als auch für unser tägliches Leben. Mit dem neuen Angebot von Informationen geht aber auch eine steigende Überlastung einher. Nur wenige der angebotenen Informationen sind für uns interessant und diese gehen vielfach in der Informationsflut unter. Deshalb brauchen wir Software-Tools, die uns beim Herausfiltern von für uns relevanten Informationen unterstützen.

Das Information Retrieval stellt hierzu viele Basistechniken zur Verfügung. Ein Großteil der Informationen auf dem WWW sind als Textdokumente gespeichert. Mit Methoden des Text Retrievals kann in Suchmaschinen wie Lycos, Altavista oder Excite in Dokumenten nach Schlüsselwörtern gesucht werden. Obwohl mit diesen Suchmaschinen bereits einige Erfolge bei der Informationssuche erzielt werden können, haben sie einige grundlegende Probleme.

Kein WWW-Katalog ist vollständig: Durch die hohe Dynamik des WWW kommen ständig neue Dokumente und Informationsquellen hinzu, während alte wegfallen.

Es wird nur nach Stichworten gesucht, nicht nach Inhalten: Der Typ der Seite, auf der ein Schlüsselwort vorkommt, wird durch Suchmaschinen nicht unterschieden (z. B. Personal Home Page oder veraltete Seite über eine Konferenz von vor 3 Jahren). Desweiteren ist oft die Semantik einer Anfrage dem Benutzer unklar.

Heutzutage werden diese Probleme meist noch manuell behandelt. Angenommen wir suchen die Personal Home Page einer Person. Zuerst probiert man mehrere Suchmaschinen durch, bis man den Namen der Person findet. Dann geht man manuell die Liste der Treffer durch, bis man endlich eine vielversprechende Seite findet. Oftmals ist diese Seite dann noch nicht die Personal Home Page. Aber vielleicht ist es eine Seite über ein Projekt, an welchem die Person arbeitet. Von dort gelangen wir dann endlich über einen Hyperlink zur gewünschten Home Page.

Wenn es mit der Suche nicht so reibungslos klappt, werden wir vielleicht noch andere Strategien anwenden. Zum Beispiel können wir mit einem der EMail-Adreßbücher die EMail-Adresse der Person herausfinden. An der EMail-Adresse können wir dann den Arbeitgeber bzw. Internet-Provider der Person ablesen, was uns dann wahrscheinlich zu ihrer Home Page führt. Unzählige andere Strategien, welche die vielen auf dem WWW verfügbaren Informationsdienste benutzen, sind ebenfalls denkbar.

In der hier vorgestellten PG soll diese Art von Informationssuche automatisiert werden. Hierbei soll ein *agentenbasierter* Ansatz verwirklicht werden. Jeder Agent ist ein Experte für bestimmte Informationsobjekte, nach denen er auf Befehl des Benutzers sucht. Dabei sollen Agenten komplexe Strategien anwenden können, wie sie sonst auch von einem Menschen benutzt würden. Mögliche Informationsobjekte sind z. B.

- Personal Home Pages
- Projekte
- Firmen
- Produkte
- Nachrichten
- Publikationen

Agenten können vielfältige Informationsdienste und auch sich gegenseitig benutzen. Um z. B. eine Publikation zu finden, kann man zuerst die Personal Home Page des Autors finden lassen. Mit Methoden der Textklassifikation erkennt ein Agent, wann er die gewünschte Seite gefunden hat.

Zur Entwicklung solcher Agenten ist es notwendig, verschiedene Techniken und Methoden zu integrieren. Dies ist eine der zentralen Herausforderungen dieses Projekts. Wir werden zeigen, wie

- Textklassifikation,
- Maschinelles Lernen,
- Planen und
- Wissenrepräsentation

in einem Agenten kooperativ eingesetzt werden können.

1.2 Definition des Projektziels

Aufbauend auf der Seminarphase zu beginn der PG, hat die Projektgruppe drei alternative Suchszenarien erstellt. Mit Hilfe diese Szenarien wurden dann die endgültigen Projektziele festgelegt.

Das Ziel ist die Entwicklung einer Agenten–Shell (BotIshelly), die modular aufgebaut ist. Es handelt sich dabei um eine Bibliothek, die die notwendigen Grundoperationen zur effizienten Implementierung von Internet–Agenten (Softbot) für die Informationssuche zur Verfügung stellt.

Modularität soll insbesondere bedeuten, daß verschiedene Ansätze z. B. zur Planung, Wissensrepräsentation und Klassifikation eingebaut werden können, ohne daß zusätzliche Änderungen in anderen Teilen der Bibliothek notwendig sind. Außerdem soll sie Nebenläufigkeit, Mehrbenutzerfähigkeit und Any-Time-Algorithmen unterstützen. Eine anpaßbare Benutzerschnittstelle sowie eine automatische Datenverwaltung sind vorgesehen.

Die Shell soll aber nicht für bestimmte Sachgebiete konfiguriert sein.

Zusammenfassend ergeben sich also folgende Anforderungen an BotIshelly als Ziel der Projektgruppe:

1. Modularität

2. Domänenunabhängigkeit
3. Anpassungsfähigkeit an ein sich schnell veränderndes Web
4. Verwirklichung intelligenter Suchstrategien
5. inhaltliche Bewertung der Suchergebnisse
6. Bereitstellung aller Grundbausteine, um konkrete Agenten mit wenig Aufwand zu realisieren
7. Integration verschiedener Methoden der künstlichen Intelligenz in einer einzigen Bibliothek, die darüberhinaus auch Module für Internetzugriffe und für die Verwaltung von im Mehrbenutzerbetrieb laufenden Agenten bereitstellt.

Um die Funktionalität der Shell zu demonstrieren und sie experimentell zu evaluieren, werden wir basierend auf BotIshelly einen Agenten prototypisch implementieren.

Kapitel 2

Seminarphase

2.1 Die Seminarfahrt

Am Anfang der PG war eine Phase geplant, in der durch Vorträge die Grundlagen für die spätere Arbeit geschaffen werden sollten. Es wurden die folgenden Themen vergeben:

- Was ist ein Agent?
- Softbots
- Einführung in das Information Retrieval
- Textklassifikation und maschinelles Lernen
- Ausgewählte Metasuchmaschinen
- Internetprogrammierung
- Klassisches Planen
- Nichtklassisches Planen
- Reinforcement Learning
- CiteSeer und Cora[1]
- Wissensrepräsentation: Frames

Jeder (der studierenden) PG–Teilnehmer hatte zu einem der Themen einen Vortrag vorzubereiten. Die schriftlichen Ausarbeitungen zu diesen sind im Anhang zu finden.

Die Vorträge wurden auf einem Kompaktseminar gehalten, das am 9. und 10. April 1999 in der Jugendherberge in Wetter Eshorn stattfand. Diese Seminarfahrt sollte auch zum gegeseitigen Kennenlernen dienen. Am Ende dieses Kompaktseminars stand der Beginn der Diskussion über die Ziele der PG. Ein wesentlicher Teil dieser Diskussion bestand darin, zu klären, was realistisch bzw. machbar erscheint und was nicht.

Die Diskussion wurde in den zweimal wöchentlich stattfindenden PG–Treffen fortgeführt. Die Ergebnisse sind im nächsten Abschnitt beschrieben.

[1]als Beispiele für domänenspezifische Suchagenten im Internet – hier Literatursuche

2.2 Ergebnisse der Seminarphase

Im Verlauf der Seminarphase hatten sich folgende Aspekte herauskristallisiert:

1. Die Agenten–Shell soll im Sinne eines Baukastens modular aufgebaut und erweiterbar sein. Die Shell selbst soll domänenunabhängig sein, aber die Realisierung von domänenspezifischen Agenten unterstützen. Die Shell soll einen "Retrieval–Ansatz", Module zu Planung und Wissen sowie Möglichkeiten zur Bewertung durch den Benutzer zur Verfügung stellen. Unter den "Retrieval–Ansatz" fallen folgende Anforderungen:

   - Informationen suchen/finden
   - Ähnliches finden
   - andere Dienste benutzen
   - Synonyme berücksichtigen

   Die Bewertung durch den Benutzer könnte durch Relevanzprüfung/Feedback erfolgen. Zur Unterstützung der Benutzer könnten auch Benutzerprofile erstellt werden.

   Zu den Modulen zu Planung und Wissen gibt es folgende Überlegungen:

   - Das Verhältnis zwischen festen Strategien und Planung muß geklärt werden. Bei der Konstruktion der Planungskomponente ist auf ein vernünftiges Verhältnis zwischen Flexibilität und Aufwand zu achten. Eine "Explosion" des Planungsaufwandes muß vermieden werden.
   - Möglichkeiten, um Strategien (Pläne) zu lernen, sollen erforscht werden.
   - Domänenbezogenes Sachbereichswissen muß in die Planung einfließen.

2. Der konkrete Agent, der mit Hilfe der Shell am Ende der PG erstellt wird, soll für einen Anwendungsbereich sein, der durch existierende Agenten (noch) nicht (so stark) abgedeckt ist. Denkbar wäre ein Agent für Software–Suchen oder für integrierte Suchen in Universitätsdomänen. Zudem soll er anpassungsfähig an ein sich schnell veränderndes Web sein. Für den Benutzer sollte er einfach und ohne große Erklärungen zu bedienen sein. Zu klären ist der Grad der Spezialisierung und die Geschwindigkeit des Agenten. Zur Diskussion stehen Online–, Offline– und Anytime–Algorithmen.

Im weiteren müssen wir immer zwischen **Agentenbenutzer** und **Shellbenutzer** unterscheiden. Der **Agentenbenutzer** ist der Anwender, der über eine graphische Schnittstelle später unseren Agenten nutzen können wird. Der **Shellbenutzer** ist der Anwender, der unsere Bibliothek nutzt, um sich einen eigenen Agenten zusammenzubauen.

Nach ausgiebiger Diskussion darüber, welche Komponenten wir auf jeden Fall für unsere Agenten–Shell brauchen, ergibt sich folgende Gliederung:

- **WRK** kürzt Wissensrepräsentationskomponente ab. Diese umfaßt die Planungsoperatoren und das Domänenwissen, das sich in Faktenwissen und Zustandswissen aufteilt.
- **Planer**: Wie werden Daten beschafft und welche Suchstrategien/Pläne werden verfolgt?
- Die **I/O** ist die Ein-/Ausgabeschnittstelle für den Agentenbenutzer. Sie hat die Aufgabe, Eingaben und Feedback des Agentenbenutzers entgegenzunehmen, Nachfragen und Ergebnisse des Agenten auszugeben sowie Debug– und Statistikausgaben (Logs) zu erzeugen.
- Die **Datenbeschaffung** ist für das Besorgen von Informationen aus dem Netz zuständig. Dies soll per HTTP, FTP und Anfragen an Suchmaschinen und andere Agenten erfolgen.
- Die **Datenanalyse** enthält die Klassifikatoren, z. B. Textklassifikatoren, und Utilities.

Bei dem Versuch, den Datenfluß zwischen den einzelnen Komponenten zu modellieren, fällt auf, daß eine zentrale Steuereinheit fehlt. Der Einbau der Steuereinheit in das Modell sowie deren Bestandteile (Ablaufkontrolle, Planausführung etc.) sind nicht offensichtlich. Es stellt sich die Frage, ob eine eigenständige Ablaufkontrolle notwendig und gewünscht ist, oder aber die verschiedenen Module autonom und gleichberechtigt miteinander kommunizieren sollen.

Um unsere grobe Aufteilung in Module zu konkretisieren (z. B. Klassen, Methoden, Datenfluß), sollen fiktive Suchen, die Softbots ausführen könnten, durchgespielt werden. Diese Szenarien sind in I 3.1 beschrieben.

Kapitel 3

Modellierungsphase

3.1 Konkretisierung durch Agentenszenarien

Da nach der Seminarphase — wie schon im vorigen Abschnitt erwähnt — zwar klar schien, welche einzelnen Komponenten für die Shell benötigt werden würden, aber nicht, wie der Kontrollfluß zwischen diesen in einem späteren Agenten aussehen könnte, wurde beschlossen, einige fiktive Suchen durchzuspielen. Es wurden drei Kleingruppen gebildet. Jede sollte sich überlegen, wie ein Agent, der eine der drei folgenden Aufgaben zu bewältigen hat, vorgehen könnte:

- Software–Suche
- Büchersuche
- Homepage–Suche

Die dabei entstandenen Agentenszenarien und die Ergebnisse bzw. Erkenntnisse, die wir aus diesen für die weitere Modellierung gewonnen haben, sind in den nächsten Abschnitten beschrieben.

3.1.1 Softwaresuche

Entwurfsbeschreibung

Systemkontrolle Die Systemkontrolle verwaltet den Gesamtablauf des Suchens. Sie stößt zunächst die Benutzungsschnittstelle an, um der Benutzerin die Möglichkeit zu geben eine Suchanfrage einzutippen. Danach stößt die Systemkontrolle den Planer an, damit er einen Plan entwickeln kann, der das Suchproblem lösen soll. Ist ein solcher Plan vom Planer zurückgeliefert worden, wird der Plan an den Planausführer übergeben.

Nun existieren zwei denkbare Möglichkeiten:

1. Die Systemkontrolle wartet auf das Ergebnis des Planausführers und stößt den Planer im Falle eines Fehlschlages erneut an oder präsentiert der Benutzerin das Ergebnis.
2. Die Systemkontrolle stößt in jedem Fall den Planer nochmals an, um evtl. einen besseren Plan zu erhalten. Dieser Plan kann z.B. besser werden, weil dem Planer mehr Zeit/Suchraum zur Verfügung gestellt wird oder weil der zweite Plan schon mit Erkenntnissen aus der Planausführung des ersten Plans arbeiten kann. Auch die Planausführung eines zweiten Plans eröffnet im Wesentlichen zwei Möglichkeiten:

(a) Die serielle Ausführung der erarbeiteten Pläne. Dabei werden die fertigen Pläne in der Systemkontrolle zwischengespeichert.

(b) parallele Abarbeitung der erarbeiteten Pläne in mehreren Planausführungsinstanzen.

Am Schluss der Suche schickt die Systemkontrolle der Benutzungsschnittstelle eine Nachricht, die Ergebnisse zu präsentieren.

Planer Der Planer soll eine Mischung aus klassischem und konditionalem Planer sein. Einerseits soll die Welt abgeschlossen sein, d.h. zu jedem Planungszeitpunkt wird nur mit den Fakten gearbeitet, die in der Wissensdatenbank zu finden sind, andererseits soll er aber auch Wissen um Alternativen und Beobachtungen haben. Um auf die Beobachtungen reagieren zu können, können Alternativteilpläne geplant werden, die die Planausführung gemäß der Beobachtung ausführt.

Die Operatoren, die dem Planer zur Verfügung stehen, holt er sich aus der Operatordatenbank. Zur Planung werden, neben den Fakten aus der Wissensdatenbank, ausschließlich die Vor- und Nachbedingungen der Operatoren benutzt.

Operator Ein Operator besteht im Wesentlichen aus drei Komponenten:

1. der Vorbedingung, die erfüllt sein muß, um den Operator anzuwenden.
2. der Nachbedingung, die erfüllt ist, nachdem der Operator angewendet wurde.
3. dem „Effektor", d.h. die Beschreibung der Handlung, die der Operator repräsentiert (i.A. also ein Net–Service–Aufruf oder ein Datenanalyse–Aufruf).

Die Operatoren werden in der Operatorendatenbank gespeichert.

Planausführer Der Planausführer ist ein Knecht der Suche. Er arbeitet den vorgegebenen Plan Schritt für Schritt ab. Dabei benutzt er ausschliesslich die „Effektoren" der Operatoren. Er beachtet *nicht* die Vorbedingungen der Operatoren. Sollte dadurch ein Fehler im Plan auftreten, so meldet die Planausführung diesen Fehler an die Systemkontrolle und wartet auf einen neuen Plan.

Net-Service Die Net–Service–Komponente beinhaltet die Funktionen, die es ermöglichen externe Daten zu erhalten. Das werden in unserer Anwendung zwar hauptsächlich Internetschnittstellen sein, können aber auch „lokale" Datenbanken im Intranet oder auf der Festplatte sein. Zugriffe auf agenteneigene Daten werden jedoch nicht über diese Komponente abgewickelt.

Datenanalyse Die Datenanalyse umfasst drei Gebiete: Lernen/Klassifizieren von Benutzern, Lernen/Klassifizieren der gefundenen Daten und sog. Content-Services, die die Struktur der gefundenen Daten zur Weiterverarbeitung aufbereiten sollen.

Alle drei Datenanalysegebiete arbeiten auf den nichtklassifizierten bzw. nichtbearbeiteten Daten des Datenbehälters oder falls eine mehrstufige Klassifikation vorgesehen ist, auf den vorklassifizierten Daten.

Klassifikatoren Die Klassifikatoren klassifizieren die gefundenen Daten in die vom Shellbenutzer vorgegebenen Klassifikationsklassen. Die Diskriminanten für die Klassifikation sollen von den Klassifikatoren erlernt werden. In die Klassifikation kann je nach Klassifikator ein Benutzerprofil eingehen.

Begriffslerner Dieser Lerner soll es nach einiger (Trainigs–)Zeit ermöglichen die Suchanfragen durch sachverwandte Schlüsselwörter (die gelernt werden sollen) zu unterstützen.

Relevance-Feedback-Lerner Dieser Lerner soll einerseits der Benutzerprofilerstellung dienen, aber andererseits auch den Begriffslerner(I 3.1.1) unterstützen, indem „wichtige“ Seiten als zum Thema passend angesehen werden und daraus sachverwandte Begriffe extrahiert werden.

Content Service Diese Datenanalysestufe soll die geholten Webseiten parsen, die Links extrahieren, sowie andere Vorarbeiten zur weiteren Analyse der gefundenen Daten erledigen.

Wissensrepräsentationskomponente Die Wissensrepräsentationskomponente gliedert sich in die Operatorendatenbank, die Wissensdatenbank, die Benutzerprofile und den Datenbehälter.

Operatorendatenbank In der Operatorendatenbank sind die Operatoren (I 3.1.1) für das Planen und die Planausführung abgelegt.

Wissensdatenbank Die Wissensdatenbank soll das Faktenwissen der Suchmaschine beinhalten. Diese Fakten müssen so abgelegt werden, dass die Klassifikatoren, der Planer und die Planausführung effizient auf sie zugreifen können.

Benutzerprofile Die Benutzerprofile, die dabei helfen sollen, dass der Benutzerin auch das präsentiert wird, was sie/er wirklich sucht, werden hier abgelegt.

Datenbehälter Der Datenbehälter soll als Dateneimer dienen. Hier legen alle Komponenten ihre Daten ab, damit sie von den Datenanalysekomponenten in Wissen überführt werden können.

Benutzungsschnittstelle Die Benutzungsschnittstelle dient zur Kommunikation mit der Benutzerin. Relevance–Feedback–Daten werden durch diese Komponente genauso gewonnen wie die eigentliche Anfrage und die Präsentationsmöglichkeit der gefundenen Seiten.

konkretes Implementierungsbeispiel

Interface (Benutzungsschnittstelle) Über das nachfolgend skizzierte graphische Interface interagiert die Benutzerin mit dem Softwareagenten. Die drei wesentlichen Elemente des Interface sind

- eine Maske mit mehreren Attribut-Wert Eingabefeldern zur Charakterisierung der gesuchten Software:[1]

Produkt	____________
Version	> *1.0*
Firma	____________
Lizenz	____________
Domäne	*Graphvisualisierung*

- ein Select-Button zur Festlegung auf einen der vorgegebenen domänenspezifischen Schwerpunkte.

Schwerpunkt	<*Info*, *Download*, *Manual*, *Tutorial*, *FAQ*> (= Klassifikationskisten)

Der N-TEXT-KLASSIFIKATOR arbeitet bzgl. dieser Schwerpunkte. Ihre relative Gewichtung bestimmt die Ergebnispräsentation (vgl. Ausführungen zum genannten Operator weiter unten).

[1]Die konkreten Feldbelegungen und Unterstreichungen stellen die Eingabe für die Beispielsuche in Abschnitt I 3.1.1 dar.

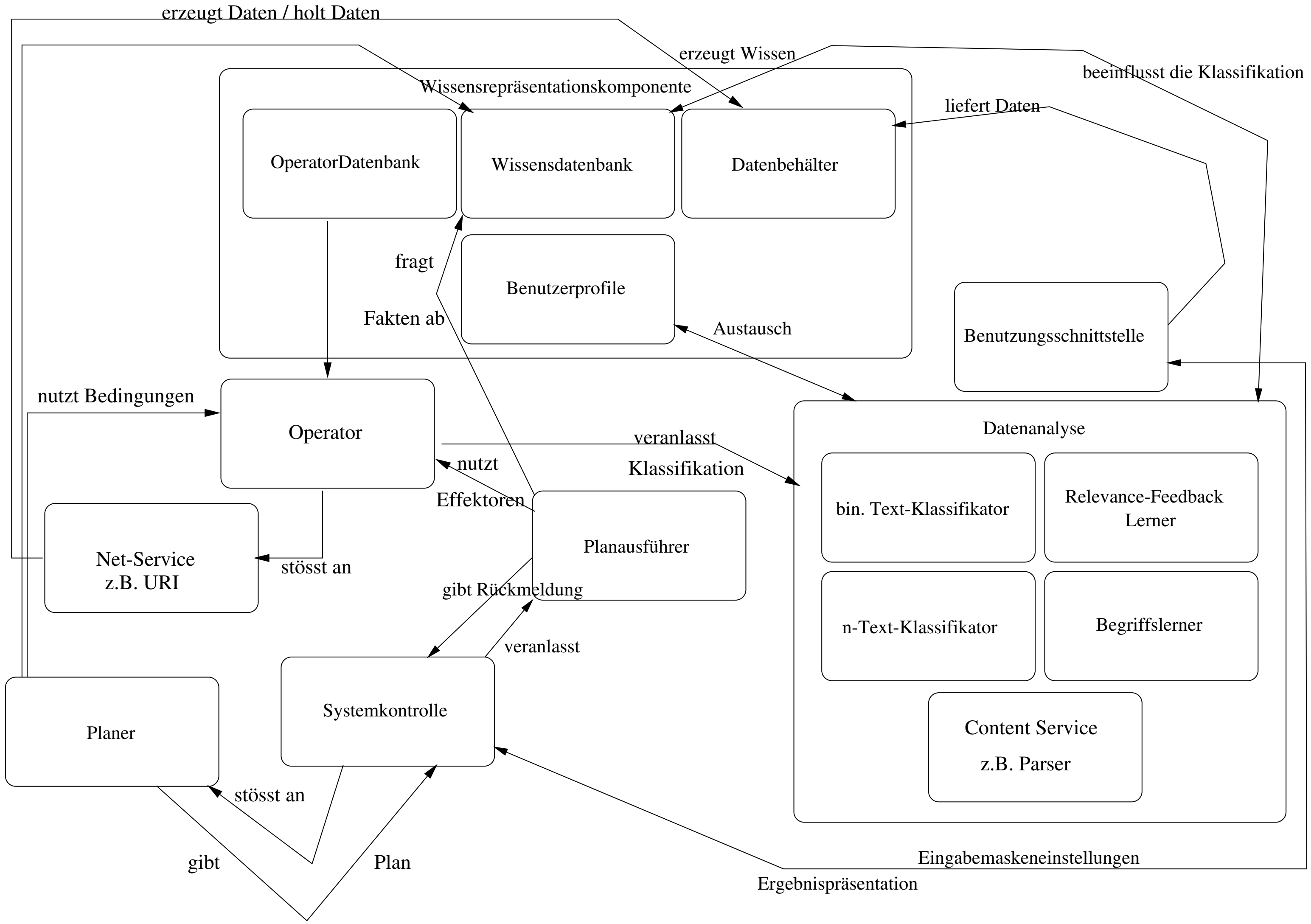


Abbildung 3.1: Entwurfsskizze

- ein Select-Button zur Vorgabe der maximalen Suchzeit. Sinnvollere Werte ergeben sich vielleicht durch den Agenten-Einsatz in der Praxis.

  Zeitangabe *<3 Min, 10 Min, jüngstes Gericht >*

Planungsoperatoren Damit der Planer *immer* einen initialen Teilplan erstellen kann, muß sichergestellt sein, daß nach Eingabe von Daten über das Interface wenigstens ein Operator anwendbar ist. Dies sichern die folgenden beiden Festlegungen:

1. Sobald über das Interface Daten D eingegeben werden, gilt
   { *rohdaten(D)* } hier etwa
   { *rohdaten([version=>1.0, domaene=graphvisualisierung, schwerpunkt=download, zeitangabe=10min])* }

2. Ausserdem ist vorgegeben:
   { *machine(www.softcrawler.com)* }
   { *machine(www.metacrawler.com)* }

Zwei Operatoren dienen der Datenbeschaffung,

- der eine realisiert Anfragen an Suchmaschinen,

  { *rohdaten(D), machine(M)* }
  SUCHANFRAGE(M,D)
  { *suchergebnis(E)* }

- der andere füllt, wenn möglich, nichtbelegte Attribut-Wert Felder des Interface über den Abgleich mit einer lokalen Software-Datenbank auf.

  { *not(feld_belegt(F))* }
  LOKALES_SW_FILE(F)
  { *feld_belegt(F) XOR not(feld(belegt(F)))* }

Operatoren zur Analyse eingeganger Daten sind:

- Ein Operator zur Auswertung der Inferface-Eingaben.

  { *rohdaten(D)* }
  MASKENAUSWERTER(D)
  { *gewichtung_der_klassi_kisten(G), feld_belegt(F) for some F* }

- Ein Linkchecker, der referenzierte Dokumente auf ihre Existenz überprüft.

  { *suchergebnis(E)* }
  LINKCHECKER(E)
  { *treffermenge(T)* }

- Ein binärer Klassifikator, der eine Menge von Dokumenten bzgl. der Belegung *eines* Attribut-Wert Feldes klassifiziert.

  { *treffermenge(T), T =/= leer, feld_belegt(F)* }
  BIN_KLASSIFIKATOR(T,F)
  { *klassifizierte_treffermenge(K), T == leer* }

- Ein technisch notwendiger Klassifikator, der Fakten so konvertiert, dass Operatoren rekursiv hintereinander anwendbar sind (bei geeigneter Strategie zur Vermeidung von Endlosschleifen).

  { *klassifizierte_treffermenge(K), K =/= leer* }
  PSEUDO_KLASSIFIKATOR(K)
  { *treffermenge(K)* }

- Ein Klassifikator, der eine Menge von Dokumenten bzgl. der vorgegebenen domänenspezifischen Schwerpunkte sortiert. Alle nichtpassenden übrigen Dokumente landen in einem speziellen Topf *T_ELSE*, der (bei Bedarf und Zeit) mit Lernverfahren genauer analysiert werden kann.

{ *treffermenge(T), T =/= leer, gewichtung_der_klassi_kisten(G)* }
N_TEXT_KLASSIFIKATOR(T,G)
{ *getypte_treffermenge(T_Info, T_Download, ..., T_FAQ, T_ELSE), T == leer* }

Die Schwerpunkte sind über eine anfangs vorgegebe, später durch Lernen verschiebbare Gewichtung relativ zueinander verknüpft. Sie soll angeben, mit welchen Wahrscheinlichkeiten klassifizierte Seiten zu dem gewünschten Schwerpunkt führen.

Bsp.: Gewählter Schwerpunkt sei *Download* und $G = [0.5, 1, 0.2, 0.2, 0.5]$ die Gewichtung. Solange die Maximalzahl der zu präsentierenden Ergebnislinks nicht erreicht ist, sollen zuerst alle Download-Seiten (1), dann alle Info- und FAQ-Seiten (0.5) und zuletzt auch Manual- und Tutorialseiten (0.2) vorgeschlagen werden.

Beispielsuche

Die folgende Sequenz aus nummerierten Fakten und fettgedruckten Operatoranwendungen illustriert eine kleine Beispielsuche zur vorher beschriebenen Implementierung.

Eingabedaten sind die Beispielwerte aus Abschnitt I 3.1.1.

(1) rohdaten([version=>1.0, domaene=graphvisualisierung,
schwerpunkt=download, zeitangabe=10min])
(2) machine(www.metacrawler.com)
SUCHANFRAGE((1), (2))
(3) suchergebnis(ref_HTML_Doc1)
MASKENAUSWERTER((1))
(4) gewichtung(0.5, 1, 0.2, 0.2, 0.5)
(5) feld_belegt(domaene), feld_belegt(version)
LINKCHECKER((3))
(6) treffermenge([url1, url2,...,urln])
BIN_KLASSIFIKATOR((6), domaene)
(7) klassifizierte_treffermenge([url2, url5, url8, url17, url20, url34])
N_TEXT_KLASSIFIKATOR((7), (4))
(8) getypte_treffermenge([url17], [url5, url20], [], [url34], [], [url2, url8])

Wenn maximal drei Ergebnislinks präsentiert werden sollen, so sind dies url5, url20 (beides Downloadseiten mit Gewicht 1) und url17 (Infoseite hat höheres Gewicht 0.5 als die Tutorialseite url34 mit Gewicht 0.2).

Ablaufsteuerung ohne Plan-Komponente ?

Da die Plan-Erstellung unter Prolog stattfinden soll, diese Anbindung jedoch relativ schwierig (und daher wahrscheinlich auch langsam) erscheint, haben wir noch einen Vorschlag erarbeitet, das Ziel auch ohne Plan zu erreichen.

Grundidee Grundlage der Repräsentation ist eine relationale Datenbank. In der realen Implementierung wird man allerdings auf ein RDBMS[2] verzichten, da auf Persistenz kein Wert gelegt wird. Außerdem wäre der Geschwindigkeitsvorteil dieser Lösung wohl nicht mehr existent.

Neben der Datenbank benötigen wird noch eine Menge von Operatoren. Diese Operatoren bestehen jeweils aus Vorbedingung und Aktion:

[2] **R**elationales-**D**aten**B**ank-**M**anagment-**S**ystem

Vorbedingung
Operator

Die Vorbedingung gibt an, auf welche Zeilen der relationalen Datenbank der Operator angewendet werden darf.

Beispieloperatoren Ein möglicher Operator wäre z.B. *VerifyLink*, der auf Zeilen der Datenbank angewendet wird, in deren verified-Spalte eine Null steht:

URLS(url, verified, classified): verified = 0
VerifyLink

Ein weiterer Operator: *ClassifyLink*. Dieser kann nur auf schon verifizierte URLs angewendet werden. Damit keine URL mehrfach klassifiziert wird, bedeutet die Belegung classified=0, daß die URL unklassifiziert ist:

URLS(url, verified, classified): verified = 1, classified=0
ClassifyLink

Ein konkretes Beispiel Es liegen drei URLs vor: zwei sind noch nicht verifiziert, eine URL ist noch nicht klassifiziert:

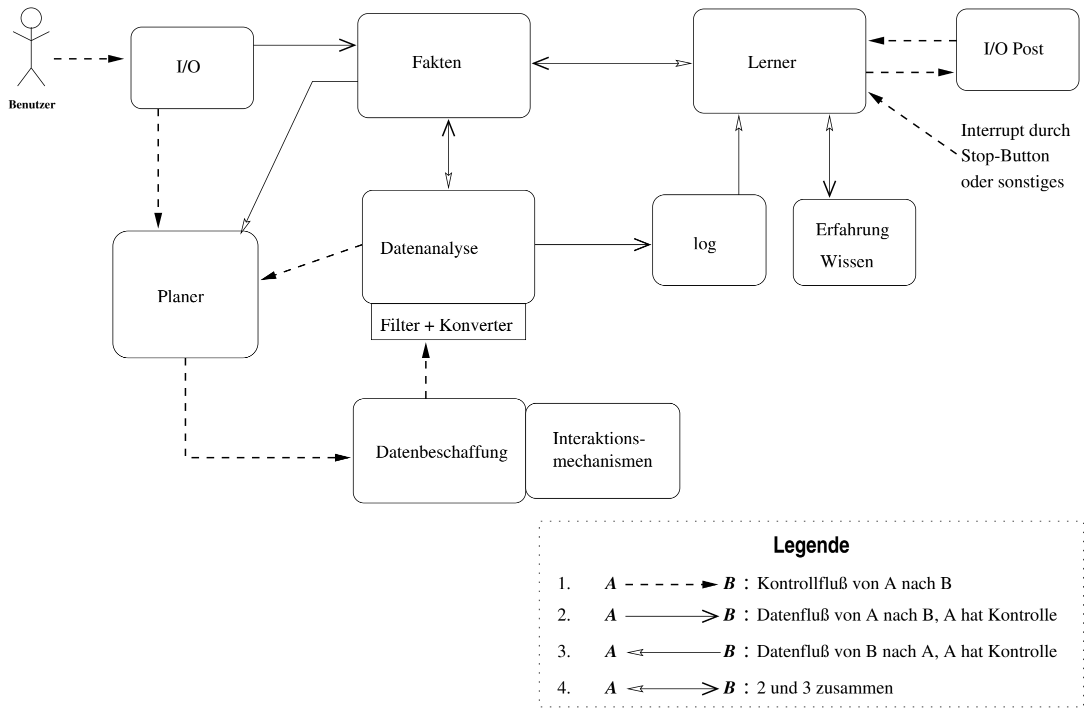
Tabelle: URLS		
URL	verified	classified
http://www.vh1.de/	0	0
http://www.sat1.de/	1	0
http://www.vivazwei.de/	0	0

Es stellt sich allerdings das Problem, welcher Operator nun angewendet wird. Auf Zeilen 1 und 3 kann *VerifyLink*, auf Zeile 2 *ClassifyLink* angewendet werden. Da kein Plan existiert, müssen gewisse Regeln definiert werden, nach denen Operatoren ausgewählt werden, da i.d.R. mehrere Operatoren auf die Zeilen der Datenbank angewendet werden können.

3.1.2 Buchagent

Die Komponenten und deren Zusammenspiel

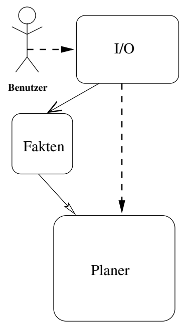
Aus welchen Komponenten der Buch–Suche–Agent besteht und wie diese zusammenarbeiten, wird durch die Grafik veranschaulicht:


Benutzer
I/O
Fakten
Lerner
I/O Post
Interrupt durch
Stop-Button
oder sonstiges
Datenanalyse
Filter + Konverter
log
Erfahrung
Wissen
Planer
Datenbeschaffung
Interaktions-
mechanismen
Legende
1. A B : Kontrollfluß von A nach B
2. A B : Datenfluß von A nach B, A hat Kontrolle
3. A B : Datenfluß von B nach A, A hat Kontrolle
4. A B : 2 und 3 zusammen

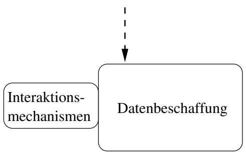
Primär-Zyklus (Such-Zyklus)

Die eigentlich Suche, die vom Benutzer angestoßen wird, ist hiermit gemeint. Folgender Ablauf ist bei einer Anfrage vorgesehen:

Der Benutzer stellt die Suchanfrage, d.h. er füllt Textfelder im Eingabeformular aus.

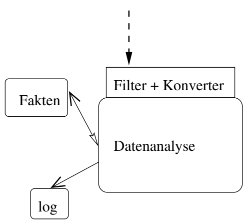


Der Planer holt die Daten und Operatoren aus den Fakten, vergleicht Vor- und Nachbedingungen, und versucht seinem Ziel (Eingabeformular zu vervollständigen) möglichst nahezukommen.



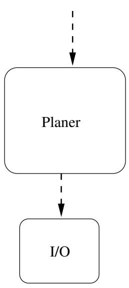
Die Datenbeschaffung delegiert die Beschaffung an spezialisierte Interaktionsmechanismen, und gibt die erhaltenen Daten weiter.

Der Filter unterscheidet die verschiedenen Formate, und schickt den Text gegebenefalls an den Konverter.



Die Datenanalyse probiert herauszufinden, ob die gefundenen Daten irgendwas mit dem Ziel zu tun haben und bewertet diese. Relevante Daten werden in das log-File geschrieben, bewertete Daten fließen in die Fakten zurück, und die Kontrolle geht wieder an den Planer, der eine neue Iteration startet, bzw.:

⋮

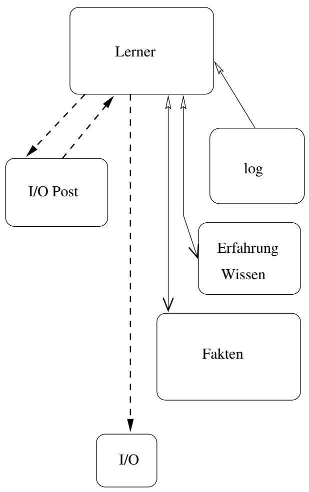


⋮

Der Planer beendet die Planung [positiv/negativ] und gibt das Ergebnis an die primäre I/O.

Sekundär-Zyklus (Lern-Zyklus)

Der Sekunderzyklus wird von dem Agenten nach Beendigung der eigentlichen, d. h. für den Nutzer "sichbaren" Suche vom Agenten gestartet, um aus der abgeschlossenen Suche "noch etwas zu lernen":



Entweder wurde ein Interupt ausgeführt oder der Planer hat den Such-Zyklus beendet, dann geht die Kontrolle entweder automatisch oder vom Benutzer veranlaßt an den Lern-Zyklus. Er fragt den Benutzer, ob er auch eine Bewertung abgeben möchte.
Der Lerner holt die Daten aus dem log-File und dem Erfahrungswissen, interpretiert diese, schreibt dies in das Erfahrungwissen zurück und versucht zusammengesetzte Operatoren zu lernen.
Die Kontrolle geht an den Primär-Zyklus zurück. Der Benutzer kann nun neue Suchanfragen stellen oder das Programm beenden.

3.1.3 Homepagesuche

Systembeschreibung

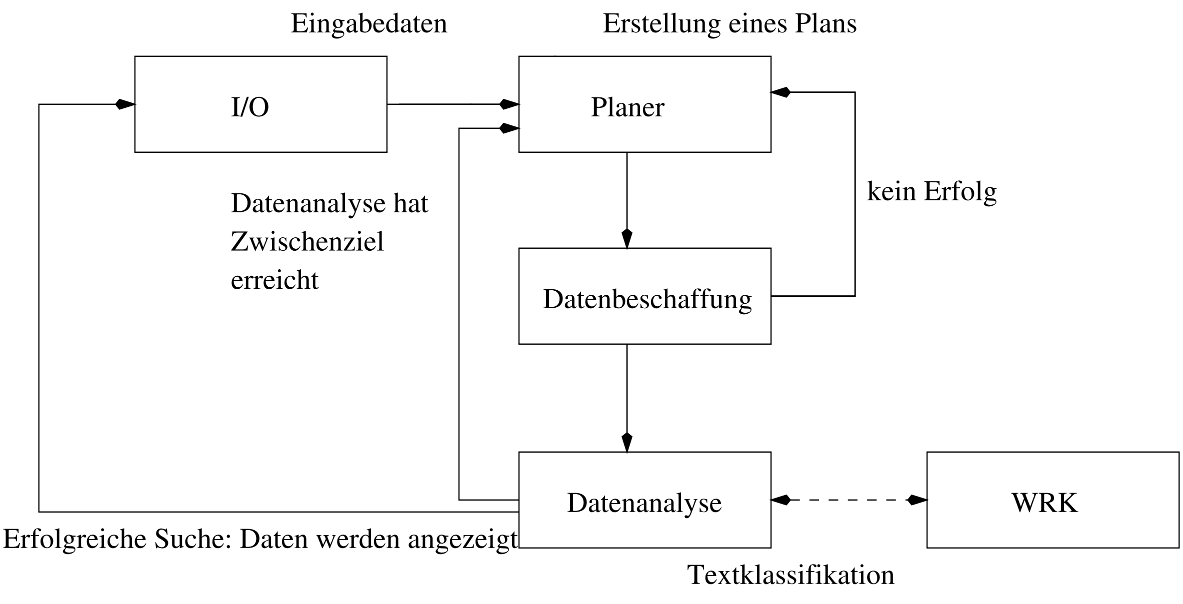


Abbildung 3.2: Systembeschreibung

I/O: Eingabemenü.	Es wird sowohl eine Suche unterstützt, die sich nur auf wenige Daten stützen kann als auch auf vollständig ausgefüllte Felder. Die Suche kann sich daher schwierigeer oder einfacher gestalten.
Planer:	Erstellung einer Sequenz (Plan), die abgearbeitet werden muß. Entscheidung über den Einsatz von Suchmaschinen, Agenten etc.
DB: Datenbeschaffung.	Sammelt die Daten, die bei dem Einsatz von Suchmaschinen und/oder Agenten gefunden wurden.
DA: Datenanalyse.	Überprüft die Daten aus der Datenbeschaffung auf ihre Relevanz für die gestellte Aufgabe.
WRK:	Unterstützt die Datenanalyse bei der Textklassifikation mittels erworbener Kenntnisse und dem Wissen aus ihren Datenbanken.

Wir schränken unsere Suche auf den englischsprachigen Raum ein. Desweiteren, da es sich bei einer Homepage offensichtlich um eine HTML-Seite handelt ziehen wir nur die HTML-Seiten in betracht.

Wir verzichten außerdem auf eine Kontrolleinheit, die Kontrolle wird von Komponente zu Komponente übergeben.

Szenario

Der Agentenbenutzer möchte die Homepage einer bestimmten Person finden. Er gibt die Daten, die ihm bekannt sind, also in diesem Fall nur den Namen in die Eingabemaske.

Die Daten werden an den Planer übergeben, der die Datenbeschaffung beauftragt die bekannten Suchmaschinen oder Suchagenten abzufragen. Dabei soll die Datenbeschaffung direkt die Homepage finden, und falls das nicht gelingt alles was mit diesem Namen zu tun hat.

Die gefundenen Seiten werden an die Datenanalyse weitergeleitet. Die Textklassifikation findet heraus, daß keine Homepage gefunden wurde, aber einige Seiten (Publikationen), in denen der Name erwähnt wird. Außerdem wird nach der Abfrage der WRK bekannt, daß in einigen Fällen neben diesem Namen auch Name einer bestimmten Universität zu finden ist. Die Datenanalyse gibt die gewonnene Information und eine von der WRK geratene Adresse an den Planer zurück.

Der Planer beauftragt die Datenbeschaffung die Adresse zu testen, falls das Ergebnis negativ ist, soll die Datenbeschaffung mittels Spider auf der Domain nach Namen und Personalverzeichnissen suchen. Die Vermutung an dieser Stelle ist, daß die gesuchte Person an der Universität angestellt ist. In dem Personalverzeichnis ist der Name aber nicht vorhanden, wohl aber in einigen Seiten eines Lehrstuhls.

Die Datenanalyse und die WRK erkennen einen Zusammenhang zwischen der Person und einem Land (Offensichtlich handelt es sich um einen Gastprofessor).

Der von dem Planer erstellte Plan enthält eine Suche nach allen Universitäten in diesem Land (falls es mehrere gibt), in den Universitäten soll nach den Personalverzeichnissen gesucht werden und nach der Homepage.

Die Datenanalyse erkennt die gesuchte Seite und gibt sie an die I/O–Schnittstelle zurück (s. Abb. I 3.3).

3.1.4 Analyseergebnis und endgültige Zieldefinition

Zieldefinition

Die Aufgabe der Projektgruppe besteht darin, eine Agenten–Shell zu erstellen, aus der Webagenten zusammen gebaut werden können. Unsere Shell wird eine Bibliothek sein, oder anders

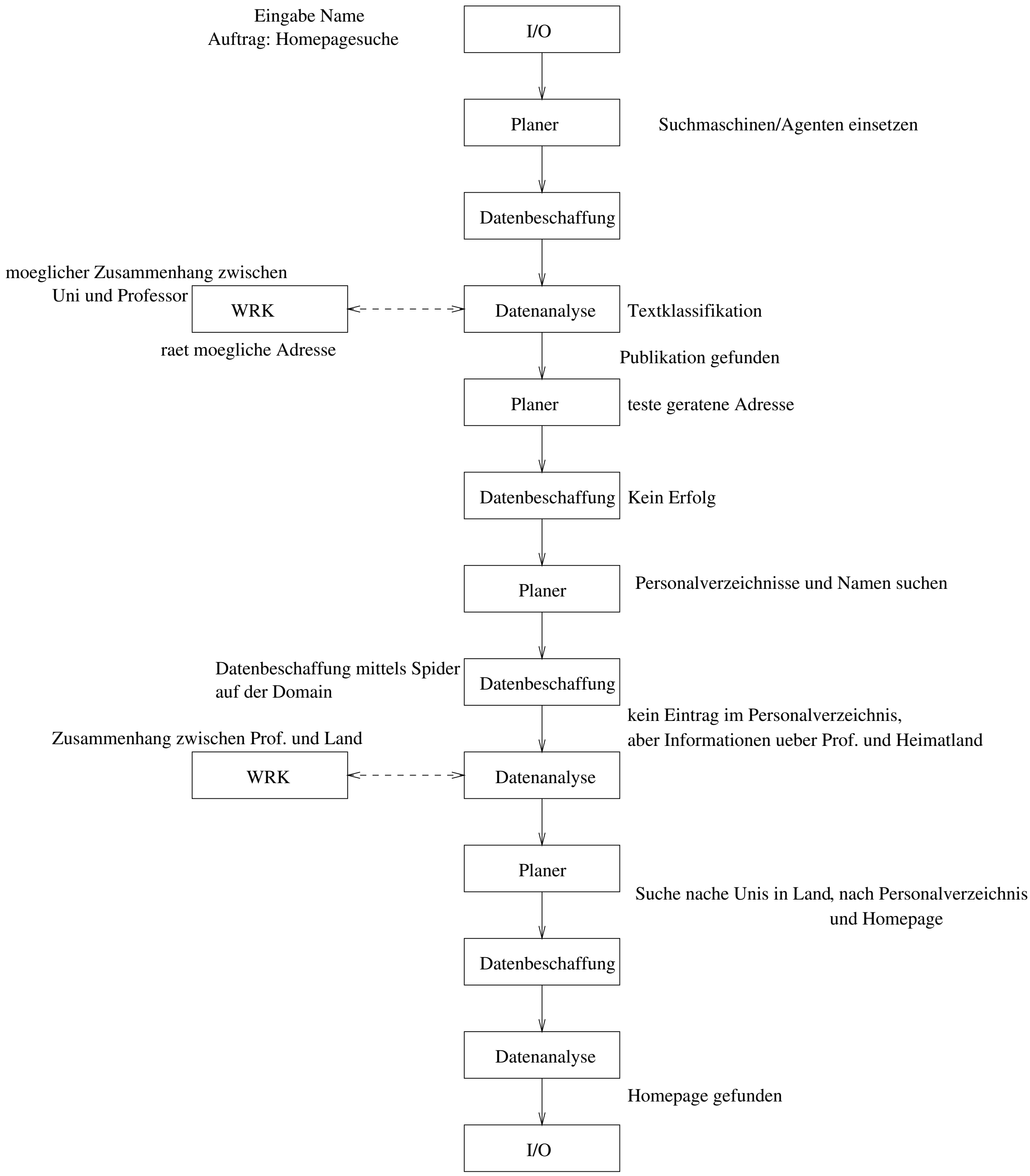


Abbildung 3.3: Ablauf einer Homepagesuche

ausgedrückt, ein abstrakter Agent, der alle notwendigen Grundoperationen hat, um (hoffentlich) erfolgreich in WWW nach Informationen(Daten) suchen zu können, der aber nicht für bestimmte Sachgebiete konfiguriert ist. Er soll variable Planungskomponenten enthalten, die es erlauben verschiedene Planungskonzepte zu verwirklichen, außerdem soll er Nebenläufigkeit, Mehrbenutzerfähigkeit und Any–Time–Algorithmen unterstützen. Auch eine automatische I/O-Erzeugung und Datenverwahltung sind vorgesehen.

Im folgenden werden wir zwischen drei Arten von Akteuren unterscheiden:

1. dem Shelldesigner
2. dem Shellbenutzer
3. dem Agentenbenutzer

Die Aufgaben des Shelldesigners sind die Planungskomponente, die Systemkontrolle und die I/O zu implementieren. Die Wissens–Repräsentations–Komponente wird nur als Hülle, die Klassifikatoren und der Lerner nur als Grundfunktionen zur Verfügung gestellt. Auch die Datenbeschaffung wird als Grundfunktion implementiert, außerdem wird eine Methodensammlung für einige Suchdienstanfragen bereitgestellt.

Der Shellbenutzer muß, um einen Agenten für ein konkretes Suchgebiet bauen zu können, problemspezifische Operatoren schreiben, die T–Box und die A–Box mit relevanten Konzepten, bzw. Instanzen füllen. Zudem muß er die Klassifikatoren trainieren und den Lerner aktivieren. Wenn der Shellbenutzer weitere Internet–Suchdienste benutzen möchte, dann muß er die Methodensammlung der Datenbeschaffung entsprechend ergänzen.

Dies alles soll nun das Glück des Agentenbenutzers vermehren, der einen Suchagenten zur Unterstützung für seine Informationssuche in WWW hat.

Beschreibung der Komponenten

I/O Die I/O ist die Schnittstelle zum Agentenbenutzer und dient der Eingabe von Suchanfragen und der Ausgabe von Ergebnissen. Das konkrete Aussehen wird je nach Aufgabe der einzelnen Agenten variieren. Im wesentlichen besteht die Schnittstelle aus einem Formular mit Eingabefeldern. Wird der Suchagent gestartet, sucht die I/O in der T–Box nach dem Startkonzept, das Informationen über die Eingabemaske, wie z. B. Größe, Position und Art der Eingabefelder, enthält und generiert daraus das Eingabefenster. Hat der Agentenbenutzer seine Suchanfrage in die Eingabefenster eingetragen, werden aus den Eingabedaten, d.h. den Strings aus den Textfeldern, eine Instanz des Startkonzeptes gebildet und in die A–Box eingetragen.

Das Suchziel muß am Anfang der Suche von dem Agentenbenutzer festgelegt werden, aber auch Daten aus den Eingabefeldern können Teilziele beschreiben. Außerdem erhält der Benutzer die Möglichkeit die aktuelle Suche abzubrechen und seine Bewertung der Ergebnisse einzugeben.

SYSTEMKONTROLLE Eine Systemkontrolle muß vorhanden sein, da es bei einer Suche nebenläufig mehrere Instanzen von Planern und Planausführung (z. B. realisiert als Threads) geben kann. Sie löst den Prozess aus und erhält die Mitteilung über den Abbruch vom Benutzer und beendet daraufhin die aktuelle Suche. Für die Any–time–Realisierung gibt es ein Signal an die Planausführungen, die daraufhin alle schon vorhandenen Ergebnisse liefern. Die Systemkontrolle sammelt die Ergebnisse der Planausführungen und entfernt doppelt vorkommende. Die Elemente der so entstandenen Liste werden nach deren Konfidenzwerten sortiert und der I/O zur Anzeige übergeben.

Zudem verwaltet die Systemkontrolle die Ressourcen und auch die Daten von mehreren, evtl. gleichzeitigen Benutzeranfragen, damit später Benutzerprofile erstellt werden können.

PLANER Planer können in zwei Ausführungen vorhanden sein: klassisch und reaktiv. Der klassische Planer definiert Teilziele, um diese zu erreichen, kann er reaktive Planer einsetzen. Der Planer wird von der Systemkontrolle initialisiert und holt sich die Daten oder die Vorbedingungen und die Nachbedingungen aus der A–Box, sucht in der OperatorenDB passende Operatoren und gibt diese Informationen zusammen mit dem erstellten Plan an die Planausführung. Die Planausführung aktiviert den Operator mit den A–Box–Parametern und der Operator schickt eine Anfrage an die Datenbeschaffung oder die Klassifikatoren. Die Anfrageergebnisse gehen über den Operator an die Planausführung zurück. Die Planausführung muß die A–Box schreiben und lesen können, um Teilergebnisse einzutragen und auszulesen. Sobald eine Instanz eines Zielkonzeptes aus der T–Box gefunden wurde, wird diese als Ergebnis an die Systemkontrolle zurückgegeben. Außerdem wird nach jedem Suchzyklus der Suchweg in das Planarchiv geschrieben.

Wissens–Repräsentations–Komponente (WRK) In der WRK wird lokales Wissen repräsentiert, aber auch das Wissen über verschiedene Suchen hinweg.

1. Konzeptionelles Wissen: T–Box
   Die T–Box soll die Suchdomäne darstellen. Dieses Wissen ist nicht lernbar, sondern muß vom Shell–Benutzer eingegeben werden. Die T–Box hat indirekten Einfluß auf das Aussehen von Operatoren. Konzepte der T–Box werden von Klassifikatoren benutzt. Die T–Box enthält ein Startkonzept, aus dem das Eingabefenster generiert wird und das mit den Benutzereingaben gefüllt, in die A–Box übertragen wird, sowie besonders ausgezeichnete (mit Flags) Zielkonzepte. Ferner gibt es ein Ausgabekonzept, das – ähnlich dem Startkonzept – Informationen über das Aussehen der Ausgabe enthält. Das Ausgabekonzept dient der I/O zur Generierung eines Ausgabefensters.

2. Instanzenwissen: A–Box
   Es existieren mehrere A–Boxen gleichzeitig, eine globale und für jede Planausführung je eine temporäre. Der Inhalt der temporären wird in das Planarchiv übertragen und steht dort dem Lerner zur Verfühgung. Der Lerner kann später zum Beispiel einzelne Adressen in die globale A–Box eintragen, so daß das Wissen der globalen A–Box von den Benutzereingaben und den bewerteten Suchergebnisen beeinflußt wird und suchenübergreifend ist.

3. OperatorenDB
   Ein Operator besteht aus Vor- und Nachbedingungen und einem Aktionsteil. Ein Operator ist anwendbar, wenn die Vorbedingungen erfüllt sind, und falls die Operation erfolgreich war, sind auch die Nachbedingungen erfüllt. Die Datenbank enthält verschiedene Operatorentypen:

   (a) atomare Operatoren

   (b) zusammengesetzte Operatoren, die einen Teilplan darstellen, bestehend aus einer oder mehreren Aktionen.

   (c) Operatoren, die die Vorausplanung unterbrechen, sie sind besonders gekennzeichnet und nur für die reaktiven Planer.

   (d) einen ausgezeichneten Anfangsoperator, der die (Such-) Ziele aus den Eingaben extrahiert.

4. Planarchiv
   Die ausgeführten und deshalb auch bewerteten Pläne werden in das Planarchiv geschrieben. Sie dienen als positive und negative Beispiele für eine nachträgliche Bewertung des Suchweges durch den Lerner.

5. KlassifikatorenDB
   Die Klassifikatorendatenbank ist eine Methodensammlung für unterschiedliche Klassifikatorentypen, zum Beispiel:

   (a) binärer Textklassifikator

(b) n-Textklassifikator

(c) Wortähnlichkeit

Der Klassifikator wird aus dieser Methodensammlung instanziiert und muß vortrainiert sein um die Daten von der Datenbeschaffung gemäß der Anfrage eines Operators bewerten zu können.

DATENBESCHAFFUNG Für alle zu unterstützende Protokolle, z. B. http, ftp, müssen jeweils eigene Methodensammlungen erstellt werden. Um Anfragen an Suchdienste wie *Altavista* stellen zu können, muß die Syntax dieser Suchdienste bekannt sein. Die Daten füllen dann den PUFFER–Datenbehälter. Außerdem teilt die Datenbeschaffung dem Operator mit, ob die Anfrage erfolgreich war oder nicht.

Der Konverter muß die Anfrageergebnisse in ein einheitliches Inhaltsformat umwandeln können. Das kann ASCII–Code sein.

DATENANALYSE Die Datenanalyse besteht aus Klassifikatoren (wie oben beschrieben) und Utilities, wie zum Beispiel aus einem Vektorisierer, der Daten passend für die Klassifikatoren aufbereitet, oder einem eigenen Lerner, der durch nachlernen die Klassifikation verfeinert. Eine Datenstruktur zur Verwaltung von Beispielen, um einzelne Klassifikatoren zu trainieren, und evtl ein Lexikon für Synonyme ergänzen die Utilities.

Die Aufgabe des Klassifikators ist es, die Daten, die von der Datenbeschaffung in den Puffer geschrieben wurden, Begriffen zuzuordnen, um für die Operatoren die Anfrageergebnisse zu bewerten.

LERNER Der Lerner kann sich nach einer abgeschlossenen Suche alle Daten dieser Suche aus dem Planarchiv holen und über die Nacht lernen. Dabei dienen die Suchpläne als positive und negative Beispiele, Ziel ist es gute Suchpläne zu lernen, erfolgversprechende Reihenfolgen von Operatoren heraus zu finden und diese dann zu komplexen Operatoren zusammenzubauen. Diese neuen Operatoren werden in die OperatorenDB eingetragen.

Als Bewertungsgrundlage dienen Kriterien wie Zeit und Erfolg der Suche, aber auch die Bewertung über die Qualität der Ergebnisse, Vollständigkeit bzw. Unvollständigkeit durch den Agentenbenutzer.

Die sich daraus ergebende Architektur ist auch in Form einer Grafik gegeben (s. Abb. I 3.4), in der daneben noch die Benutzungsabhängigkeiten zwischen den einzelnen Teilen angedeutet sind.

3.2 Ergebnisse/Konzepte der Kleingruppenmodellierung

Um die einzelnen Komponenten der Shell, wie sie im vorherigen Abschnitt beschrieben wurden, zu implementieren, wurden Kleingruppen gebildet. Eine jede hatte sich mit einer Komponente auseinanderzusetzen, was zunächst durch die Modellierung der jeweiligen Komponente mit *Rational Rose* (vgl. I 4.1) geschehen sollte. Die sich aus diesen Modellierungen ergebenden Konzepte werden in den nachfolgenden Abschnitten komponentenweise beschrieben.

3.2.1 Systemkontrolle/IO

Start einer Suche

Die vorgegebene IO in der Shell wird durch Java Servlets realisiert. Um eine Suche zu starten, wird eine URL aufgerufen, die eine Methode in der Klasse `InputmaskServlet` startet. InputmaskServlet ist dafür zuständig, die Eingabemaske aufzubauen, und eine Methode in der Systemkontrolle aufzurufen, die weitere Initialisierungen vornimmt. Es ist jedoch für den Shellbenutzer möglich, die IO beliebig umzuschreiben.

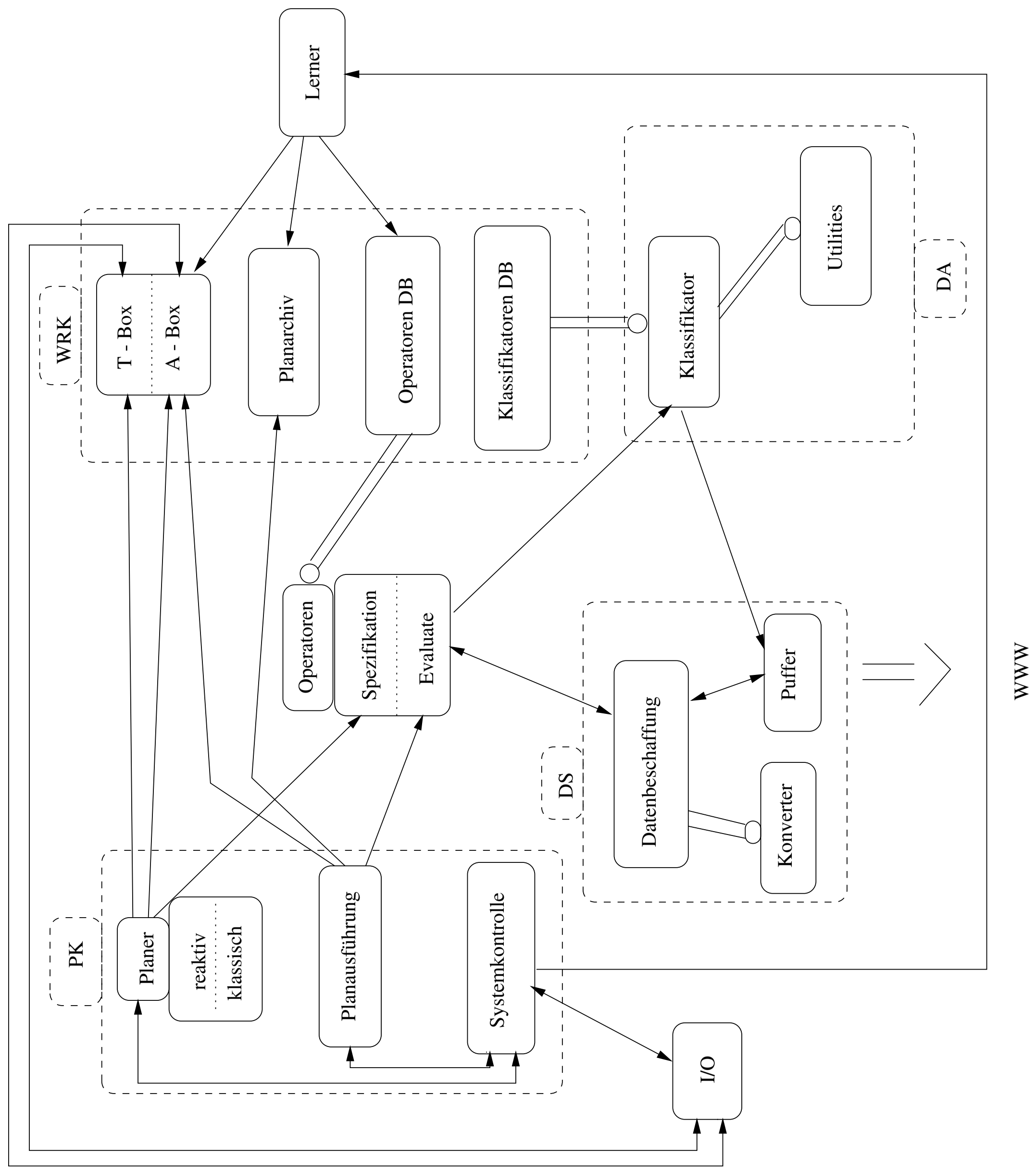


Abbildung 3.4: Architektur, die sich aus der Analyse der Agentenszenarien ergab

Initialisierung

Die Systemkontrolle instanziiert zunächst einige Klassen. Dazu gehören `SearchID`, `PlanID`, `Results`, `ResultsList`, `Reference_Table`, `StartConcept`, `Input` und `Output`. `Search ID` ist eine globale ID, die aus Datum und Uhrzeit generiert wird.

Da mehrere Suchanfragen parallel ablaufen können und für jede neue Suche eine Systemkontrolle instanziiert wird, gibt es für jede Suche eine `SearchID`, die diese Instanzen unterscheidet. Innerhalb einer Suche kann es mehrere Planer geben, um diese zu unterscheiden gibt es die `PlanID`. Die Klasse `Results` sammelt die Ergebnisse aller Planer und filtert identische Strings heraus. `ResultsList` ist eine LinkedList aller Resultsinstanzen. Die Ergebnisausgabe basiert auf `ResultsList`. Die `Reference_table` ist eine Liste aller Planinformationsobjekte, die in einer Suche vorkommen. Die Klasse `Input` enthält die Eingabedaten, generiert die Eingabemaske, holt die Eingabedaten und registriert Abbruchkommandos durch den Benutzer. `Output` generiert die Ausgabemaske und schreibt die Ergebnisse hinein. Ausserdem wird die globale `A-Box` und `T-Box` geladen.

Aufbau der Eingabemaske

Input holt aus der `T-Box` das Eingabekonzept und kann damit die Eingabemaske generieren und auf dem Bildschirm ausgeben. Der Benutzer kann nun die Maske ausfüllen. Nach Druck auf den Submit-Button holt sich die Systemkontrolle die Referenz auf die Eingabedaten. Aus diesen Eingabedaten wird das `Startkonzept` ausgefüllt. Eine Kopie der globalen `A-Box` wird erstellt und das ausgefüllte `Startkonzept` in die Kopie geschrieben.

Suchzyklus

Für jeden Planertyp wird höchstens ein Planer erzeugt. Für jeden Planer, der erstellt wird, wird ein Planinformationsobjekt instanziiert, das unter anderem die Referenz auf eine Kopie der `A-Box` mit dem Startkonzept enthält. Der Planer wird als Thread gestartet und ihm wird eine Referenz auf ein Planinformationsobjekt übergeben. Der Datenaustausch zwischen Systemkontrolle, Planern und Planausführern findet immer über Planinformationsobjekte statt.

Werden mehrere Planer für einen Agenten implementiert, werden diese auch parallel gestartet. Dies wird durch Threads realisiert.

An dieser Stelle wird die Event-Loop in der Systemkontrolle gestartet, die Ereignisse von Planern, Planausführern und von der IO verarbeitet, daß heisst den Nachrichtenaustausch zwischen diesen Komponenten verwaltet. Ein Event kann von den entsprechenden Objekten ausgelöst werden, und dieses Event wird dann in der Event-Loop verarbeitet.

Ein Planer kann ein Event setzen, welches der Systemkontrolle mitteilt, daß ein Plan oder ein Teilplan fertiggestellt ist. Beim Auslösen des Events wird das entsprechende PlanInformationsobjekt untersucht: Ist das Attribut PlanFinished auf TRUE gesetzt, ist der erstellte Plan der letzte Teilplan, d.h. nach der erfolgreichen Ausführung liegen die Ergebnisse vor. Sonst ist nur ein Teilplan erstellt.

Hat eine Planausführung erfolgreich einen Teilplan ausgeführt, löst die sie ein Event aus, das der Systemkontrolle dises mitteilt. Es werden die Zwischenergebnisse, auf die über das Planinformationobjekt zugegriffen werden kann, wieder zum entsprechenden Planer weitergeleitet. Dieser plant dann mit den Ergebnissen einen weiteren Teilplan oder den Endplan.

Wird ein Plan erzeugt, der das Endergebnis liefern soll, wird dieser auch zur Planausführung geschickt. Sobald diese Instanz der Planausführung fertig ist, liegen Ergebnisse vor, die zunächst im Result-Objekt zwischengespeichert werden. Dort werden sie mit anderen Ergebnissen der anderen Planausführer zusammengeführt.

Sind alle Planausführer mit ihren endgültigen Plänen fertig, weiß die Systemkontrolle, daß nun die Ergebnisse auf dem Bildschirm angezeigt werden müssen. Sie holt sich aus der `T-Box` (Referenz aus Planinformationsobjekt) das Ausgabekonzept, daß das Aussehen der Ausgabemaske enthält. Darauf wird eine Methode in `Output` aufgerufen und dabei ein Link auf die Results übergeben, die die Ausgabemaske zusammen mit den Ergebnissen generiert. Diese werden dann auf dem Bildschirm ausgegeben.

Weitere Features von Systemkontrolle/IO

Es ist außerdem möglich, sich Teilergebnisse einer Suche auszugeben, während diese noch läuft. Laufende Planer und Planausführungen werden dabei nicht unterbrochen. Es werden lediglich die aktuell vorhanden Teilergebnisse aller Planausführungen in die Results geschrieben und auf dem Bildschirm ausgegeben.

Zudem gibt es noch einen Abort-Button, wird dieser betätigt, werden sofort alle Planer und Planausführer gestoppt, und der Agent wird beendet. Es wird nichts weiter ausgegeben.

Bei der Ausgabe der Ergebnisse sorgt die Systemkontrolle dafür, das doppelt vorkommende URLs eliminiert werden.

Zudem verwaltet die Systemkontrolle die Ressourcen und auch die Daten von mehreren, evtl. gleichzeitigen Benutzern, damit später Benutzerprofile erstellt werden können. Dieses Feature wird aus Zeitgründen jedoch nicht implementiert.

3.2.2 Wissensrepräsentation

Für die Wissensrepräsentation wird ein Termsubsumtionsformalismus mit einer T– und einer A–Box verwendet. Die T–Box ist der terminologische Teil, in der die Begriffe definiert und in eine Begriffsstruktur eingeordnet werden. Die T–Box stellt das Domänenwissen (z. B. über die Struktur des WWW) des Suchagenten dar.

Termini der T–Box:

Definition: Ein **Konzept** repräsentiert einen Begriff, der in eine Begriffshierachie eingebettet ist. *Anything* ist der allgemeinste Begriff, von dem sich alle anderen Begriffe ableiten lassen.

Definition: Eine ***isA*–Beziehungen** ist keine Rolle im eigentlichen Sinne, sondern stellt die Relation zwischen Ober- und Unterkonzept dar.

Definition: Eine **Rolle** ist eine binäre Relation zwischen Konzepten.

Die T–Box wird als Graph modelliert, indem die Konzepte als dessen Knoten und die möglichen Rollen zwischen den Objekten als dessen Kanten aufgefaßt werden. Außerdem gibt es eine weitere, ausgezeichnete Kantenmenge, die durch die *isA*–Beziehungen zwischen den Konzepten gegeben ist. Diese erzeugt eine Hierarchie der Konzepte, da die *isA*–Kanten zusammen mit den Konzepten einen zyklenfreien Graph bilden. In diesem ist `anything` — unter welches jedes Konzept subsummiert wird — die Wurzel.

Auf die Java–Klassen `Concept` und `Role` kann nicht von außen (d. h. von außerhalb des Paketes) zugegriffen werden. Die Klasse `T-Box` stellt die Methoden zur Verfügung, mit denen man Konzepte und Rollen finden und auslesen kann. Es existiert ein Subsumtionstest, der beliebige Konzepte `a` und `b` darauf überprüft, ob gilt: `a` subsumiert `b`. Ferner gibt es Methoden, um Konzepte und Rollen als Ziele der Suche zu markieren. Außerdem kann die T–Box gespeichert, geladen und kopiert werden. Das Kopieren ist notwendig, weil für jede Suche eine eigene T–Box erzeugt werden muß, in der die Suchziele dieser speziellen Suche markiert werden.

Die A–Box ist der assertionale Teil der Wissensrepräsentation, d. h. in der A–Box werden die Individuen/Instanzen und die tatsächlich bestehenden Rollen gespeichert.

Termini der A–Box:

Definition: Eine **Instanz** ist eine Entität eines Konzeptes. Instanzen können gleichzeitig zu verschiedenen Konzepten gehören.

Definition: Eine **konkrete Rolle** ist eine tatsächlich bestehende Rolle zwischen zwei Instanzen und wird häufig auch als "Rolleninstanz" bezeichnet.

Die A–Box ist eine Hashtable, die als Elemente die Instanzen enthält. Es können nur solche Instanzen in die A–Box geschrieben werden, die zu einem Konzept aus der T–Box gehören. Eine Instanz kann mehr als einem Konzept zugeordnet werden, wobei die Konzepte nicht in Ober-/Unterkonzept Relation zueinander stehen müssen. Jede Instanz hat eine Liste von Konzepten, denen sie zugeordnet wird. Außerdem hat jede Instanz eine Liste mit den konkreten Rollen, die zu dieser Instanz existieren. Es können ähnlich wie bei den Konzepten nur dann konkrete Rollen angelegt werden, wenn es in der T–Box eine Rolle mit diesem Namen gibt und sowohl die Startinstanz, als auch die Zielinstanz in ihrer Konzeptliste die Konzepte haben, zwischen denen die Rolle in der T–Box definiert ist.

Neben einer eindeutigen Id–Nummer und dem Namen der Instanz, der nicht eindeutig sein muß, kann ein dazugehörender Inhalt vom Typ `NetResult` gespeichert werden. Zusätzlich hat jede Instanz noch weitere Attribute, wie z. B. `Instance.date`, das das Datum der Eintragung festhält, `Instance.counter`, das angibt, wie oft eine Instanz zur Planerstellung benutzt wurde, oder `Instance.delete`, das für den Planer oder den Lerner wichtig ist.

Auch auf die Klassen `Instance` und `ConcreteRole` kann nur über die A–Box zugegriffen werden. Die Klasse `A-Box` enthält die Methoden, um Instanzen manipulieren zu können. Es existieren Methoden, um Instanzen und konkrete Rollen als Benutzereingaben zu markieren. Zudem kann auch die A–Box gespeichert, geladen und kopiert werden.

Genau wie bei der T–Box wird es mehrere A–Boxen gleichzeitig geben, darunter eine globale, die über alle Suchen persistent ist und die vor der Initialisierung des Suchagenten mit Inhalt gefüllt werden muß. Für jede Suchanfrage wird eine Kopie der globale A–Box erzeugt — "suchglobale A–Box" genannt —, in der die Instanzen, die durch die Anfrage hinzugekommen sind, markiert werden. Für jeden Planer wird wiederum eine Kopie der suchglobale A–Box angelegt, die durch einen Planausführer mit neu dazukommenden Instanzen angefüllt wird. Ist eine Suche abgeschlossen, überträgt anschließend der Instanzenlerner die neu hinzugekommenen Instanzen, für die es sinnvoll erscheint, in die globale A–Box.

3.2.3 Operatoren und Operatorendatenbank

Operatoren haben die Aufgabe, eine Eingabe entgegenzunehmen (die Vorbedingung) und einzig aus diesen Informationen eine Ausgabe zu generieren (die Nachbedingung(en)). Sie werden von einem Planer zu einem Plan zusammengefaßt, das ist ein gerichteter Graph, in dem die Operatoren die Knoten sind, und die Richtung der Kanten die Ausführungsreihenfolge der Operatoren bestimmt. Der Planausführer schließlich arbeitet den Plan ab und bringt die Operatoren zur Ausführung.

Die zur Verfügung stehenden Operatoren werden in einer **Operatorendatenbank** zusammengefaßt. Diese Datenbank soll außer den üblichen Verwaltungsfunktionen Planer bei ihrer Arbeit unterstützen können. Dabei sind Routinen denkbar, die z. B. alle Operatoren mit der gleichen Vorbedingung selektieren.

Attribute der Operatoren

Ein Operator hat auf jeden Fall folgende Attribute:

- die Vorbedingung
- die Nachbedingungen
- die Gewichtung
- den Index der erfüllten Nachbedingung
- eine eindeutige ID
- einen Typ
- einen Namen

Die einzige Vorbedingung eines Operators wird durch eine Liste von Prädikaten realisiert, wobei letztere als konjunktiv miteinander verknüpft interpretiert werden. Prädikate können mehrstellig sein und werden realisiert durch eine Liste von Prädikatargumenten. Diese Argumente wiederum bestehen aus einem Namen, einem T–Box–Konzept und einer A–Box–Instanz.

Ein Operator kann mehrere Nachbedingungen haben, hat aber mindestens eine. Wir nehmen an, daß nur genau eine Nachbedingung nach der Ausführung gültig sein kann; welche das ist, entscheidet der Operator während der Ausführung. Eine Nachbedingung besteht aus einer Add– und einer Delete–Liste, die an die aus STRIPS–Planern bekannten Listen angelehnt sind.

Beispiel: Ein Operator, der zu einer Zahl `a` vom Konzept `INT` eine Zahl `b` vom Konzept `INT` addiert, das Ergebis dann in `c:INT` speichert und `b` löscht, hat als Vorbedingung (nicht instantiiert) `{(a,INT,_), (b,INT,_)}` und als Nachbedingung `ADD{(c,INT,_)}, DEL{(b,INT,_)}`. Nicht in den Nachbedingungen auftretende Prädikatargumente bleiben implizit erhalten.

Methoden

Neben den üblichen `get`– und `set`–Methoden für die Attribute, enthalten Operatoren eine `eval`–Methode, die vom Planausführer benutzt wird, um den Operator auszuführen. Von der geschickten Implementierung der `eval`–Methode eines Operators wird es abhängen, wie erfolgreich ein Operator seine Aufgabe erfüllen kann.

3.2.4 Planinformationsobjekte

Planinformationsobjekte stellen Informationspakete dar, die zwischen Planern, Planausführung, Planarchiv und Systemkontrolle hin– und hergereicht werden. Diese Objekte enthalten die Attribute `searchID` und `planID`, die die Suche und den Plan eindeutig kennzeichnen sollen. Um die Eindeutigkeit auch über Agenteninstanzen hinweg garantieren zu können, böte sich ein Datumstring, vielleicht in Kombination mit einem Rechnernamen an. Zusätzlich beinhalten die Planinformationsobjekte die drei Flags `planFinished`, `planSuccess` und `backToSender`. Die ersten beiden Flags sind zur Zuordnung der Planteile im Planarchiv notwendig, da verschiedene Pläne in unterschiedlichen Zeitintervallen, d. h. in unterschiedlicher Reihenfolge in das Planarchiv geschrieben werden können[1]. Das Ende eines Plans wird durch das gesetzte Flag `planFinished = true` gekennzeichnet. Um einem Lerner mitzuteilen, ob ein Plan erfolgreich ausgeführt wurde, d. h. keine Fehler während der Planausführung aufgetreten sind, wird das Flag `planSuccess =`

[1] Es können z. B: folgende Teilplanfolgen im Archiv stehen: Plan1, Plan2, Plan3, Plan2, Plan2, Plan3, Plan1

`true` gesetzt. Ist `backToSender` auf `true` gesetzt, dann soll das PlanInformation–Objekt nach der Planausführung an den ursprünglichen Planer zurückgegeben werden.

Weiterhin enthält ein Planinformationsobjekt die lokale A–Box und die T–Box, deren suchspezifischen Zielkonzepte markiert werden, den Plan, den der Planausführer ausführen bzw. der in das Planarchiv geschrieben werden soll, die Startkonzeptvariable `startConcept`, aus der die Zielextraktion erfolgt, und ein Array `instancesToSearch`, in das die „Suchinstanzen" aus dem Startkonzept eingetragen werden.

Planobjekte, die in das Planarchiv eingetragen werden sollen, unterscheiden sich inhaltlich von Planobjekten, die zwischen Planausführer und Planer ausgetauscht werden. Während die Objekte zwischen Planer und Planausführer den kompletten Planbaum und die komplette lokale A–Box beinhalten, enthält das Objekt, welches ins Planarchiv eingetragen werden soll, nur die Instanzen in der lokalen A–Box, die während der Ausführung neu hinzukamen.

Zur Zielextraktion wird die Methode `goalExtraction` eines Planinformationsobjektes aufgerufen, die aus dem Startkonzept die zu kennzeichnenden Zielkonzepte heraussucht und markiert und die „Suchinstanzen" extrahiert. Am Ende der Suche kann die Methode `verify` aufgerufen werden, um die Instanzen aus den, unter die Zielkonzepte fallenden Instanzen herauszusuchen, die tatsächlich mit der Zielinstanz gemeint waren[2].

3.2.5 Planer

Das Analyseergebnis der Großgruppe sieht für die Planerkomponente zwei Planertypen vor: den sog. *reaktiven* und den sog. *klassischen* Planer. Der klassische Planer soll zum Erreichen von Teilzielen reaktive Planer verwenden können — implizit ist damit ein dritten Planertyp eingeführt, ein *hybrider* Planer.

Die Planergruppe hat zunächst Einsetzbarkeit und Kombinierbarkeit dieser (implizit und explizit) im Analyseergebnis genannten Planertypen genauer untersucht, (siehe die folgenden Abschnitte zu den einzelnen Planertypen). Da sich hierbei nur der reaktive Planertyp als unproblematisch erwies, beschafften wir uns Literatur über Erweiterungsmöglichkeiten klassischer Planer und existierender Implementierungen in dieser Richtung.

Fazit der Planergruppe ist, daß kein bekanntes Planungskonzept zur Verfügung steht, daß für unsere Zwecke direkt übernommen werden kann. Dieser Eindruck wurde durch das Auffinden des DFG Projekts *WebPlan* (siehe Planer Typ D) und seiner Ziele untermauert. Die Planergruppe versucht nun, auf der Basis von ihr bekannten erweiterten klassischen Planern, einen geeigneten Planer zu entwickeln, siehe weiter unten, Planer Typ C.

Aus dem bisher Gesagten ergibt sich, daß zu diesem Zeitpunkt nicht absehbar ist, ob dieser Planer bzgl. Laufzeit und Planqualität akzeptable Ergebnisse liefern wird und auch nicht, wie es um Eigenschaften wie Korrektheit, Vollständigkeit und Termination steht!

An dieser Stelle sei darauf hingewiesen, daß die *Operatordefinitionen* immens wichtig für den (schnellen) Planungserfolg sind. Wenn die Operatoren nicht geschickt definiert sind, dann kann auch der beste Planer nichts mehr retten. So kann eine ungeschickte Operatordefinition z. B. zu einer doppelt exponentiellen Laufzeit in der Anzahl der Operatoren führen.

Unser Ziel, *Any–Time–Algorithmen* zu verwirklichen, ist unserer Meinung nach zu einer Interpretationsfrage geworden. Der Agentenbenutzer kann jederzeit nach Ergebnissen fragen, aber es müssen keine vorhanden sein, weil die vorherige und aktuelle Suche noch keine fand. Es wäre auch möglich, als Initialplan z. B. immer die Suchmaschine *Nathan* zu fragen und deren Ergebnis zunächst zu verwenden.

[2] z. B. aus allen Druckerseiten die gefunden wurden, nur die HP520–Druckerseiten herausfiltern

Rein klassische Planer

Das sind Planer, die streng nach dem Prinzip des Planers STRIPS funktionieren. Dieser Planertyp ist — in seinem strengen Sinne — aus folgenden Gründen verworfen worden:

- Unsere *Objektwelt* ist *dynamisch*. Mit jeder neuen herangeschafften URL erweitert sich unsere Objektwelt.
- Unser Planer braucht *sensorische Effekte*, um Wissen über die Welt zu bekommen. Der Planer benutzt z. B. einen Klassifikator, um zwischen zwei möglichen Welten zu unterscheiden.

Rein reaktiver Planer — Planer Typ A

Das scheint ein sehr unproblematischer Planertyp zu sein, da er „nicht intelligent“ ist und nur Pläne der Tiefe 1 erstellt.

Planer Typ B

Dieser Planer ist ein Hybridplaner, der einen Planungsbaum aufstellt und soweit plant, wie er kann ohne daß er eine Entscheidung treffen muß, die evtl. nicht binär gefällt werden kann. Ist eine solche Stelle im Planungsbaum gefunden, dann sind zwei Dinge möglich:

Typ B1: Wird auf irgendeiner Ebene e des Planungsbaums eine mehr als binäre Entscheidung erwartet, dann wird ein Plan nur bis zur Stufe $e-1$ entwickelt und nach der Ausführung dieses Plans weiter geplant.

Typ B2: An einer Stelle des Planungsbaums wird eine mehr als binäre Entscheidung erwartet. Dann wird der entsprechende Ast auf der Ebene gekappt (Bildung einer Sackgasse) und es werden alternative Möglichkeiten im Baum gesucht. Erst wenn alle Zweige des Baums Sackgassen sind, wird der z. B. längste oder sinnvollste Plan ausgewählt und zur Ausführung gebracht.

Fazit: Nach längerer Diskussion kann sich die Planergruppe keinen echten Hybridplaner mehr vorstellen, da der Planer nur auf den Konzepten plant. Wenn ein beliebig weit vorausschauender Planer aber keine Operatorfolge findet, die das Zielkonzept wahr macht, so kann ein reaktiver Planer doch auch keinen Operator hinzuerfinden — es sei denn, es gäbe einen Operator, der wirklich nur für den reaktiven Planer zur Verfügung stünde; aber auch von dieser Möglichkeit hat die Gruppe Abstand genommen.

Planer Typ C

Da STRIPS beschränkte Planer für unsere Zwecke nicht geeignet sind, wurde in der Literatur nach Erweiterungen klassischer Planer Ausschau gehalten.

Bei den klassischen Planern hat sich offenbar *GraphPlan (GP)* [Blum and Furst, 1997] als vorerst schnellster Planer durchgesetzt. Konditionale Erweiterungen werden von [Koehler et al., 1997] (IP^2, C–Implementierung) und [Anderson et al., 1998] vorgeschlagen, die Behandlung von Ungewißheit bzgl. initialer Bedingungen zeigt [Smith and Weld, 1998]. Zusätzliche sensorische Effekte werden von [Weld et al., 1998] (*SGP*, LISP–Implementierung) eingebaut. Dynamische Objektwelten behandelt jedoch keiner der genannten Planer.

Da *SGP* unserer Problematik noch am ehesten gerecht wird, soll sich der Versuch, einen geeigneten Planertyp zu entwickeln, an *SGP* orientieren. Zum aktuellen Zeitpunkt werden von unserer Gruppe folgende Abweichungen vom Standard *(S)GP* für nötig gehalten:

Virtuelle Objekte Um dynamisch hinzukommende Objekte (URLs) in unserer Welt modellieren zu können, abstrahieren wir pro Suchmaschinenanfrage i von der konkreten Anzahl der URL–Treffer und repräsentieren jeweils alle neuen Objekte durch ein einziges Objekt $\_\_x_i$. Für den Planer ist dies ein gewöhnliches Objekt, dessen „Auftauchen in der Welt" wir zum Planungszeitpunkt bestimmen und indizieren können. Der Planausführer interpretiert dieses Objekt wegen seiner Markierung als Variable für Instanzen eines Konzepts und kann zur Planausführungszeit heuristisch entscheiden, welche (wieviele) konkrete Konzeptinstanzen er für diese Variable einsetzen will.

SearchHistoryTable Es erscheint sehr schwierig, über geeignete Operatorbedingungen identische Suchmaschinenfragen im Plangraphen zu vermeiden. Als einfache Lösung außerhalb der Operatorbedingungen bietet sich die Verwaltung einer Tabelle an, die pro Planer protokolliert, welche Suchmaschinen schon mit welchen Suchwörtern angefragt wurden. Die Anwendbarkeit eines Operators kann dann durch den Abgleich mit dieser Tabelle zusätzlich eingeschränkt werden. Wir erhoffen uns durch diese Maßnahme einen Performanzgewinn.

Anreicherung der P–Level mit generelleren Propositionen Das T–Box–Wissen sollte beim Planen ausgenutzt werden. Eine Möglichkeit ist, in jedem Proposition–Level für jede neu dort eingetrage Propositon $p(x)$ auch alle generelleren Propositionen $q(x)$ einzutragen — Voraussetzung ist also, daß q p subsumiert (sinnvollerweise nur für $q \neq ANYTHING$). Dadurch können Operatoren, die spezielle Begriffe in allgemeinere Begriffe umwandeln, entfallen. Das bedeutet, daß die Pläne kürzer werden, und der Planer schneller fertig wird.

Um die Explosion des Planungsaufwandes einzugrenzen, können aufgrund der globalen A–Box Zweige, die voraussetzen, daß keine Instanzen gefunden worden sind, abgeschnitten werden. Wenn die globale A–Box nämlich schon Instanzen des Konzepts enthält, dann ist von vornherein klar, daß auch nach Ausführen eines Operators noch welche vorhanden sein werden. Der Fall, daß keine vorhanden sind, braucht daher nicht mehr betrachtet werden. Je voller die globale A–Box ist, desto schneller kann geplant werden.

Planer Typ D

Der vom DFG–Projekt *WebPlan* [Hüllen et al., 1998] eingesetzte Planer *CaPlan* unterstützt zum gegenwärtigen Zeitpunkt noch keine sensorischen Effekte und ist damit für uns (noch) nicht nutzbar.

3.2.6 Planausführung

Die Planausführung entpackt aus dem `Planinformation`–Objekt den zu verarbeitenden Plan. Diejenigen Operatoren, deren Vorbedingungen noch mit Instanzen gefüllt werden müssen, werden vom Planausführer belegt. Sollte das nicht möglich sein, schlägt die Planausführung fehl.

Sollte der Planausführer mehrere Instanzen finden, mit denen er die Operatorvorbedingungen belegen kann, so wird eine Heuristik eine Instanz auswählen, mit der die Verarbeitung weitergeführt wird. An dieser Stelle soll evtl. ein Backtracking ermöglicht werden, so daß eine Vollständigkeit (bzgl. der A–Box) garantiert werden könnte. Um Ressourcenprobleme zu vermeiden, sollte die Anzahl der verfolgten Instanzen jedoch beschränkt sein. Die `maxInstances`–Variable wird mit dem Konstruktor gesetzt und von der Systemkontrolle bestimmt.

Die anzuwendene Heuristik soll eine Methode des Planausführers sein, die den Einsatz der Heuristik transparent macht und die Änderung der Heuristik vereinfacht.

Kann ein Operator in der OperatorDB nicht gefunden werden, wird die `OperatorNotFoundException` geworfen. Scheitert die Ausführung eines Operators, wird die `OperatorFailedException` geworfen.

Dadurch sollte aber im Falle mehrerer Instanzen nur die aktuelle Instanzverfolgung abgebrochen werden. Falls die Planausführung scheitern sollte, z. B. weil Vorbedingungen der Operatoren nicht erfüllt werden können, löst der Planausführer das Event `planExecuteAbortedEvent` aus. Trotzdem kann vom Planausführer ein `PlanInformation`–Objekt angefordert werden, welches die bis zum Zeitpunkt des Abbruchs hin gefüllte A–Box enthält, so daß die Erkenntnisse der fehlgeschlagenen Ausführung gesichert werden.

Belegung der Operatorvorbedingungen

Die Belegung der noch nicht belegten Operator–Vorbedingungsprädikate erfolgt mit Hilfe der Operator–Methode `getUnverifiedPrecondPred` und der Planausführer–Methode `verifyPredicate`. Liefert `getUnverifiedPrecondPred` kein `Predicate` mehr, so sind alle alle Vorbedingungsprädikate belegt und der Operator wird ausgeführt. Solange `getUnverifiedPrecondPred` noch ein `Predicate` liefert, wird dieses zur Belegung an `verifyPredicate` übergeben. `verifyPredicate` versucht, nichtinstantiierte `PredArguments` mit A–Box–Instanzen zu belegen. Wenn keine Instanz existiert, dann scheitert der Plan.

3.2.7 Datenbeschaffung

Das Paket Datenbeschaffung stellt für den Agenten die Schnittstelle zum Netz dar. So ist es dem Agenten möglich, auf Dateien via HTTP oder FTP zuzugreifen oder Suchmaschinen zu befragen.

Für den Zugriff auf Dateien über ein bestimmtes Protokoll (z. B. HTTP) existieren Klassen (z.B. `HTTPNetService`), die Unterklassen der abstrakten Klasse `NetService` sind. Der konzeptionelle Hintergund hierfür ist eine möglichst einfache Erweiterung der Datenbeschaffung. Für jedes Protokoll (das als Unterklasse von `NetService` realisiert wird) muss u. a. die Methode `getNetResult` implementiert werden, die als Eingabe eine URL erhält, daraufhin die durch die URL referenzierte Datei herunterlädt und als Ergebnis zurückliefert. Dadurch, dass alle konkreten Net–Services Instanzen von `NetService` sind, kann bei der Nutzung von den konkreten Net–Services abstrahiert werden, indem nur die durch `NetService` definierte Schnittstelle benutzt wird. So muss schon bestehender Code nicht für jeden neuen Net–Service angepasst werden. Diese Vorgehensweise wird weiterhin durch die Klassenmethode `getNetService` unterstützt, die einen zu der übergebenen URL passenden (konkreten) Net–Service zurückgibt.

Die Ergebnisse eines Net–Services, also heruntergeladenen Dateien, müssen in Form von Instanzen bestimmter Klassen geliefert werden, die den jeweiligen Typ der Datei widerspiegeln. Die jeweilige Klasse muss Methoden bereitstellen, über die man Zugriff auf die Merkmale einer Datei erhält (z. B. Ermitteln des Titels bei HTML-Dokumenten). Natürlich gibt es Programmteile in einem Agenten, für die zwar die Existenz einer heruntegeladenen Datei von Interesse ist, nicht aber deren konkrete Ausprägung. So ist es z. B. sinnvoll, dass ein Textklassifikator sowohl auf ASCII–Texten, als auch auf HTML–Dokumenten o. Ä. arbeiten kann, ohne sich um das spezielle Textformat kümmern zu müssen. Generell kann gesagt werden, dass sich Eigenschaften (wie z. B. in obigem Beispiel die Repräsentation von textuellen Inhalten) unterschiedlicher zu repräsentierender Dateien überschneiden werden. Die hier beschriebene Situation stellt einen der Hauptgünde dar, wegen denen OOP zur Zeit so beliebt ist. So ist es auch in Java leicht möglich dies mit Hilfe von Vererbung als Hierachie von Klassen zu modellieren. Superklasse dieser Hierarchie ist die abstrakte Klasse `NetResult`, die die Eigenschaften festlegt, die allen Dateirepräsentationen gemein ist. Die folgende Grafik zeigt die Vererbungshierarchie der einzelnen Klassen:

Direkt von `NetResult` erbende Klassen sind ebenfalls abstrakt und stellen nur eine grobe Typisierung bzw. Spezialisierung des Dateityps dar. So existiert z. B. die Klasse `TextNetResult`, die die Superklasse aller textorientierten Dateien ist (HTML, ASCII etc.). Die auf dieser Ebene angesiedelten Klassen (`TextNetResult`, `AudioNetResult` etc.) stellen für die jeweilige Art von

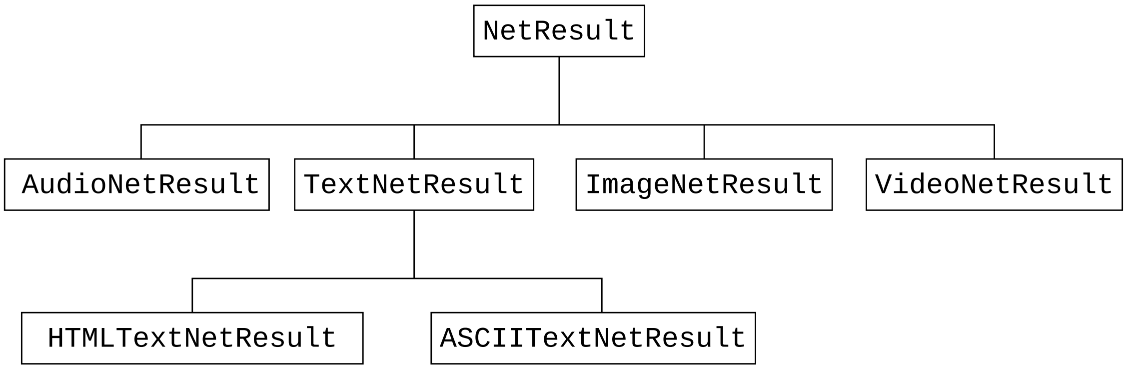


Abbildung 3.5: Hierarchie der Klassen zur Dateirepräsentation

Dateien sinnvolle Funktionen bereit (z. B. das Auslesen des Textes bei `TextNetResult`–Instanzen in gleichem Textformat).

Eine Klasse, die einem konkretem Datei- bzw. Dokumentyp entspricht (z.B. `HTMLTextNetResult`), muß dann die verschiedenen Methoden (aller) seiner abstrakten Superklassen implementieren (also z. B. das Auslesen des Textes bei HTML-Dateien durch Entfernen der HTML-Tags und die Umwandlung der Sonderzeichen). Darüber hinaus können noch spezielle Methoden vorhanden sein, die für den repräsentierten Dateityp von Nutzen sein könnten.

Die Datenbeschaffung bietet darüberhinaus das Abfragen von Suchmaschinen an. Da auch in diesem Bereich eine Erweiterung in Form des Hinzufügens von neuen Suchmaschinenschnittstellen ohne das Anpassen schon bestehenden Codes möglich sein soll, wird auch hier das schon bei den Net–Services beschriebene Abstraktionsprinzip eingesetzt. Es existiert demnach eine abstrakte Superklasse (`SearchService`), deren Methoden von konkreten Suchmaschinenanbindungen (z. B. `NathanSearchService`) implementiert werden müssen. Dabei muss in jedem Fall die Methode `doQuery` implementiert werden, die eine Menge von Wörtern als Eingabe erwartet, diese als UND–verknüpfte Suchanfrage an die jeweilige Suchmaschine weiterreicht und die Ergebnisse der Suchmaschine entsprechend aufbereitet zurückliefert.

3.2.8 Datenanalyse

Für die Entwicklung eines Konzeptes ist zunächst wichtig, die Hauptaufgabe der Datenanalyse zu bestimmen. Da es sich bei der Shell um eine Umgebung handelt die es ermöglichen soll, Suchagenten für das Internet zu konstruieren, konnte das Ziel der Analyse nur die Bewertung des Inhalts von Internet-Seiten sein. Schließlich sollte anhand dieses Textes ermittelt werden, ob eine übergebene Internet-Seite auf die Suchanfrage eines Agentenbenutzers passen könnte. So läßt sich die Datenanalyse auf die Textklassifikation beschränken.

Auf dem Gebiet der Textklassifikation gibt es schon einige Ansätze und Algorithmen, von denen vier für die Implementierung in der Shell ausgewählt wurden.

- **Bayes-Klassifikator**: Der Bayes-Klassifikator basiert auf einer Modellierung von Text mit wahrscheinlichkeitstheoretischen Methoden.

- **k-NN-Klassifikator**: Bei dem k-Nearest-Neighbor-Verfahren handelt es sich um einen Klassifikationsalgorithmus, der in der Statistik entwickelt wurde.

- **Support Vector Maschine (SVM)**: Prinzip: Structural Risc Minimization.

- **Rocchio-Klassifikator**: Der Rocchio-Algorithmus wurde als Methode zum Relevance Feedback im Rahmen des Information-Retrieval-Systems `SMART` entwickelt.

Da diese Klassifikatoren-Algorithmen in ihrer Struktur Ähnlichkeiten aufweisen, kann ein gemeinsames Grundgerüst für alle Klassifikatoren entwickelt werden . Um diesen Ziel weiter zu verfolgen, müssen drei Hilfsmittel den Klassifikatoren zur Verfügung stehen.

Als erstes Hilfsmittel wird ein `Vectorizer` benötigt, der die Vektorisierung der Dokumente übernimmt. Die Vektorisierung erhält ihre Bedeutung dadurch, daß erst durch die Umwandlung eines Dokuments in einen Zahlenwert (hier: `Documentvector`) ein Vergleich zwischen verschiedenen Dokumenten und somit letztendlich auch eine Sortierung möglich ist.

Um eine einheitliche Bearbeitung dieses `Documentvectors` zu ermöglichen, wird die Klasse `Documentvector` erzeugt. Dort sollen alle wesentlichen Methoden zur Erzeugung, Weiterverarbeitung und Berechnung von `Documentvectoren` abgelegt werden.

Als letztes der drei Hilfsmittel dienen die verschiedenen Wörterbücher oder auch `Dictionaries` genannt. Hier werden drei Arten unterschieden:

- `Feature`-Wörterbuch: Dieses Wörterbuch stellt die Grundlage für den `Documentvector`dar. Die Dimension sowie die Positionen der Einträge in den `Documentvector` hängen ganz alleine von diesem Wörterbuch ab.
- `Negativ`-Wörterbuch (Stop–Wort–Liste): Wörter aus diesem Wörterbuch werden bei der Vektorisierung nicht berücksichtigt. Der wesentliche Vorteil dieses Wörterbuchs liegt darin, unbedeutene oder häufig vorkommende Wörter unmittelbar vor der Klassifikation aus dem Text herauszufiltern.
- `Synonym`-Wörterbuch: Bei der Klassifizierung können auch verwandte Wörter eingesetzt werden.

Zum Aufbau der Klassifikatoren sei vorab gesagt, daß die Klassifikation in zwei Phasen abläuft.

1. Phase: Die erste Phase wird durch das Training bestimmt. Hier wird der Klassifikator auf seine eigentliche Aufgabe vorbereitet. Um diese Aufgabe lösen zu können, werden bereits bewertete Beispieldaten zur Verfügung gestellt, die dem Klassifikator bei der korrekten Einordnung des zu klassifizierenden Dokuments helfen sollen.

2. Phase: Die zweite Phase besteht aus der eigentlichen Klassifikation (Vorsicht!!! Es kann erst mit der Klassifikation begonnen werden, wenn die Trainingsphase abgeschlossen wurde). Hier wird anhand der gewonnenen Erkenntnisse aus dem Training der neue Text eingeordnet bzw. klassifiziert.

Alle Klassifikatoren erben von der Klasse `Classifier`, die somit das Grundgerüst für jeden Klassifikator darstellt. Dieses Gerüst zeichnet sich durch zwei wesentliche Methoden aus.

Auf der einen Seite wäre das die Methode `train()`. Sie stellt die Trainingsmethode dar. In diesem Zusammenhang stellt sich die Frage, wer oder was trainiert werden soll? Kurz gesagt, **jeder** Klassifikator muß trainiert werden. Dies funktioniert wie folgt: Da die Textklassifikation ein bestimmtes Ziel verfolgen soll, müssen zielorientiert Beispiele ausgewählt werden, die sowohl positiv wie auch negativ sein können. Wann ein Beipiel als negativ oder positiv gilt, wird durch eine Bewertung festgelegt. Ist eine geeignete Beispielmenge gefunden, so kann das Training beginnen. Während des Trainings wird das `Feature`-Wörterbuch angepaßt und die `Documentvectoren` zu jedem Dokument erzeugt. Erst wenn alle Beispieldaten das Training durchlaufen haben, kann mit der eigentlichen Klassifikation begonnen werden.

Die zweite Methode heißt `classify` und sorgt für die eigentliche Klassifikation. Von der Datenbeschaffung wird der zu klassifizierende Text vom Typ `NetEntity` übergeben. Um die Trainingsdaten und den neuen Text auf eine vergleichbare Ebene zu stellen, wird der Text in einen `Documentvector` umgewandelt. (Wichtiger Punkt an dieser Stelle: Da sich der `Documentvector` am Feature-Wörterbuch orientiert, ist eine weitere Anpassung der Wörterbücher während der

Klassifikationsphase zu unterbinden). Erst jetzt kann eine Ähnlichkeitsberechnung durchgeführt werden, die letztendlich zum Klassifikationsergebnis führt.

Da die Klassifikatoren oft eine `Documentvector`-Darstellung in TF-IDF-Form benötigen, können eventuell noch zwei Methoden den jeweiligen Klassifikatorklassen–Klassen hinzugefügt werden, die folgende Formeln berechnen:

- Nachdem alle Dokumente vektorisiert wurden, beginnt die Gewichtung über einen zweiten Faktor: Inverse Document Frequency

  $$IDF(w) = log\frac{n}{DF(w)}$$

  (n: Gesamtzahl der betrachteten Dokumente, DF: Anzahl der Dokumente, die das Wort w mindestens einmal enthalten). Um die Anzahl der Dokumente herauszufinden, die das Wort w enthalten, müssen das `Feature`–Wörterbuch und sämtliche Vektoren durchlaufen und überprüft werden. Anhand des `Feature`–Wörterbuchs kann ein Array erzeugt werden, daß die IDF–Werte zu den Wörtern abspeichert.

- TF–IDF–Gewichtung berechnen: TF steht für Term-Frequency und beschreibt, wie häufig ein Wort in einem Text vorkommt. Je häufiger ein Wort in einem Dokument vorkommt, desto wichtiger (TF–Teil) und wenn das Wort in vielen Dokumenten vorkommt desto unwichtiger kann es sein (IDF–Teil).

- Um den TF–IDF–Vektor zu erhalten muss nun die TF–Repräsentation des Textes mit der IDF–Repräsentation der Wörter aus dem `Feature`–Wörterbuch elementweise multipliziert werden.

Durch die Vererbung von der Klasse `Classifier` und den Einsatz der zugreifbaren Klassen `Dictionary`, `Vectorizer` und `Documentvector`, wird auch eine Einbindung von neuen Textklassifikatoren in die Shell möglich.

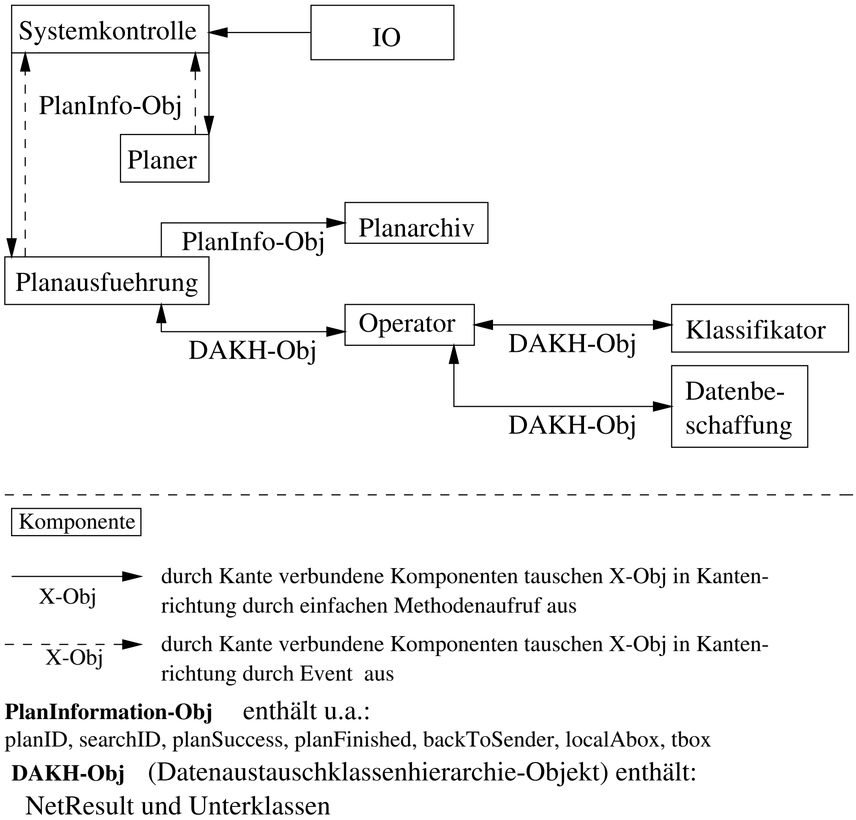
3.2.9 Zusammenfassung der Kleingruppenmodellierung

Im folgenden wird ein Gesamtbild vorgestellt werden, das das Zusammenspiel der einzelnen Komponenten verdeutlichen soll.

Nach der Initialisierung notwendiger Objekte für eine Suche wird der Benutzer zur Eingabe der Suchbegriffe aufgefordert. Nach Druck auf den Bestätigungsknopf wird die Methode `StartSearch()` in Flowcontrol aufgerufen. Dort wird ein Clone von `A-Box` und `T-Box` erstellt, die während der Suche benutzt werden. Zur Zielextraktion wird die Methode `extractGoal()` aufgerufen, die aus dem Startkonzept die zu kennzeichnenden Zielkonzepte heraussucht und die "Suchinstanzen" extrahiert.

Nun werden die zur Verfügung stehenden Planer gestartet. Jeder Planer bekommt eine Referenz auf jeweils ein Planinformationsobjekt, mit Hilfe dessen die Planer und Planausführer mit der Systemkontrolle kommunizieren. Dieses Planinformationsobjekt enthält neben der `SearchID` und `PlanID` die lokale `A-Box` und die lokale `T-Box`.

Folgendes Diagramm soll das Zusammenspiel der Komponenten der Shell etwas verdeutlichen.



Die weitere Kontrolle während eines Suchablaufs findet nun über Events statt, die Planer und Planausführer senden und die Systemkontrolle verarbeitet. Für Planer und Planausführer gibt es jeweils verschiedene Events. Die Planer können die Systemkontrolle nach Erstellung eines Plans darüber unterrichten, ob dieser Plan nur ein Teilplan ist der zunächst ausgeführt werden soll. Ist dies der Fall, wird nach der Ausführung des Teilplans wieder der Planer desselben Typs gestartet, der an diesem Plan weiterarbeitet. Zu diesem Zweck wird jedem Plan eine eindeutige `PlanID` gegeben, mit der ein Plan während einer Suche identifiziert werden kann. Eine weitere Möglichkeit ist, daß der Planer den letzten Teilplan eines Gesamtplans erstellt hat. Dann wird dies durch ein entsprechendes Event der Systemkontrolle mitgeteilt und dieser letzte Teilplan wird ausgeführt.

Ist ein Teilplan fertig gestellt, wird dieser an den Planausführer übergeben. Dieser führt den Plan mit Hilfe von Operatoren aus. Diese Operatoren laden angeforderte URLs mit Hilfe der Datenbeschaffung aus dem Netz. Ist es notwendig Seiten auf ihren Inhalt hin zu klassifizieren, werden die Klassifikatoren angesprochen. Für die Kommunikation zwischen Planausführern, Operatoren, Klassifikatoren und der Datenbeschaffung gibt es das Datenaustauschobjekt, das das Net–Result mit seinen Unterklassen enthält.

Die Planausführer können verschiedene Events an die Systemkontrolle schicken. Für den Fall, daß nur ein Teilplan ausgeführt wurde, wird der Systemkontrolle mitgeteilt, daß der Teilplan wieder an den entsprechenden Planer weitergeleitet werden soll. Eine weitere Möglichkeit ist, daß der Planausführer feststellt, daß die Suche mit diesem Plan keinen Sinn mehr macht, und er die Ausführung daraufhin abbricht. Im Planinformationsobjekt dieses Planers wird dann ein Flag gesetzt, das das Ende dieses Plans und seiner Ausführung kennzeichnet.

Andererseits kann es sein, daß die Planausführung mit dem letzten Teilplan eines Gesamtplans fertig ist, dann wird auch hier im entsprechenden Planinformationobjekt ein Flag gesetzt, das Auskunft darüber gibt, daß dieser Planer nun fertig ist, und auch nichts weiteres mehr tut.

Die Systemkontrolle überwacht diese Flags und wenn sie feststellt, daß in allen Planinformationsobjekten dieses Flag auf TRUE steht, dann weiß sie, daß die Suche endgültig beendet ist. Dies wiederum wird dann an die IO weitergegeben, die in dem Ausgabefenster dann die Endergebnisse auf dem Bildschirm anzeigt.

Über die Methode **`verify()`** kann die Systemkontrolle sich während einer laufenden Suche die bisher gefundenen Suchergebnisse in die Ergebnisliste eintragen lassen. Dafür gibt es in der Defaultbenutzeroberfläche einen Button, mit dem man sich die Zwischenergebnisse anzeigen lassen kann.

Kapitel 4

Implementierung der Bibliothek

4.1 Warum Java und Rational Rose?

Innerhalb der Seminarphase stellte sich die Projektgruppe die Frage, in welcher Programmiersprache und mit welchen Hilfsmitteln BotIshelly modelliert und entwickelt werden sollte. Als Alternativen wurden zunächst die folgenden Programmiersprachen genannt:

- C++
- Java
- Perl
- Prolog

Unklar war, ob ausschließlich eine, oder eine Kombinaton verschiedener Programmiersprachen benutzt werden sollte.

In der Seminarphase ergab sich dann, daß die Projektgruppe eine Programmbibliothek zur Verfügung stellen wollte. Um Erweiterungen gut zu unterstützen wurde einstimmig die objektorientierte Programmierung als Paradigma beschlossen. Somit fielen Perl und Prolog als primäre Alternative weg. Es blieb noch die Auswahl zwischen C++ und Java.

Für C++ sprachen die gute Performanz und die Anzahl der gleichzeitig möglichen Internetverbindungen. Java zeichnete sich vor allem durch gute Modularität und einfache Dokumentierungsmöglichkeit (Javadoc) aus. Desweiteren stellte Java (im Gegensatz zu C++) in weiten Teilen eine strukturierte Programmierung sicher.

Da die Projektgruppe eine Programmbibliothek entwickeln wollte, die einfach zu erweitern ist und nicht „das Schnellste auf dem Markt" seien musste, ergab sich so die Wahl von Java als Programmiersprache der Shell.

Nachdem Java als Programmiersprache der Shell festgelegt wurde, kam noch die sekundäre Nutzung der anderen Programmiersprachen in Frage. Nach einer Überprüfung der Einbindungsmöglichkeiten ergab sich allerdings, daß diese nur mit relativ hohem Performanzeinbußen (mittels „Piping") möglich gewesen wäre, da eine Schnittstelle wie JPL[1] leider aus technischen Gründen unter Solaris nicht zur Verfügung stand.

[1] JPL bettet eine Prolog–Engine durch Verwendung von Java Native Interface (JNI) und Prolog Foreign Language Interface(FLI) in Java ein.

So hätte die Realisierung des Planers in `Prolog` z.B. dazu geführt, daß ständig zwischen der Virtual–Machine von `Java` und `Prolog` gewechselt worden wäre. Die Konsistenzhaltung der Daten wäre ebenfalls aufwendig und ressourcenintensiv gewesen. Deshalb wurde auf eine Nutzung von anderen Programmiersprachen verzichtet.

Die Wahl von `Rational Rose` als Modellierungswerkzeug ergab sich dann aus der Wahl von `Java` als Programmiersprache. Nach einer Beratung beim STL[2] stellte sich heraus, daß `Rational Rose` zu diesem Zeitpunkt das einzige — von dem STL unterstützte — Modellierungsrogramm war, das `Java` unterstützte. Da die Projektgruppe ein grafisches Frontend zur Modellierung der direkten Klassenmodellierung vorzog wurde `Rational Rose` als Modellierungswerkzeug benutzt. Das STL kümmerte sich um die Installation von `Rational Rose`.

4.2 Änderungen/Erweiterungen der ursprünglichen Modellierung durch die Implementierung

Nachdem die Modellierung mit *Rational Rose* abgeschlossen war, wurde der Code–Rahmen erzeugt und die Kleingruppen begannen mit der Implementation ihrer jeweiligen Komponente. Die Änderungen, die sich dabei ergaben, werden in den nächsten Abschnitten komponentenweise beschrieben.

4.2.1 Systemkontrolle und IO

Die grundlegende Änderung, die bei der Implementierung gemacht wurde, bestand darin, daß die verschiedenen Suchen und die dazugehörenden Objekte nicht über ein `Session`–Objekt zwischen den verschiedenen Servlets hin- und hergereicht werden, sondern dies nun über eine Liste und Such–IDs passiert. Die Liste befindet sich in der Klasse `MainTable` und ist als ***LinkedList*** implementiert worden. Jedes Element der Liste repräsentiert eine Suche, d. h. in jedem Element befinden sich außer einer `SearchID`–Instanz noch Instanzen von `EventListener`, `ResultsList`, `PlanArchive`, `Reference_table`, `Input`, `Output`, `StartConcept` und `Flowcontrol`. Die Klasse `MainTable` selbst enthält neben der Liste auch noch die Referenzen auf die globale A–Box, die globale T–Box, die Klassifikatoren- und die Operatorendatenbank.

Die Servlets bekommen als Parameter immer die ***SearchID*** übergeben und können so in der Liste die aktuelle Suche identifizieren und die nötigen Objekte daraus auslesen.

Auch die Möglichkeit, die angefangene Suche abbrechen zu können, war in dem Entwurf nicht vorgesehen. Dafür sorgt jetzt das Abort–Servlet. Es stoppt alle zur Zeit laufenden Planer und Planausführungen und löscht das Element mit der aktuellen Such–ID aus obiger Liste. Das Abort–Servlet wird über den Abort–Button erreicht, der während eine Suche läuft angezeigt wird.

Eine weitere Änderung bezieht sich auf die Darstellungsweise von End- und Zwischenergebnissen. In der ersten Version wurden immer ein Fenster für die Endergebnisse und ein Fenster für die Zwischenergebnisse geöffnet. In der letzten Version kann der Agentenbenutzer zwischen einem und zwei Ausgabefenstern wählen. Wählt er zwei Fenster bleibt die Ausgabe wie bisher. Anderenfalls werden dem Benutzer in regelmäßigen Abständen die aktuellen Zwischenergebnisse angezeigt, bis die Suche beendet wird. Das Intervall, in dem die Aktualisierung vorgenommen wird, kann der Benutzer in der Eingabemaske einstellen.

Auch die Verwaltung der Planer hat sich geändert. Die `Reference_table` ist eine Liste von `Plan`–Objekten. Jedes `Plan`–Objekt enthält ein `PlanInformation`–Objekt, einen Planer, der mit diesem `PlanInformation`–Objekt gestartet wurde, und eine Boole'sche Variable. Diese ist `TRUE`, falls der

[2] Software-Technologie-Labor

Planer den letzten Teilplan erstellt hat und dieser ausgeführt wurde. Dadurch wird erreicht, daß das Ende der Suche eindeutig festgestellt werden kann.

Da eine Bewertung — "Ranking" auf neudeutsch — der einzelnen Ergebnisse einer Suche eingeführt wurde, wurde die interne Darstellung der einzelnen Einträge nachträglich geändert. Jeder Eintrag ist in einem `QueryResult`–Objekt gekapselt, das daneben noch die Bewertung dieses Eintrags enthält. Damit die Ergebnisse in der Ausgabemaske dem Ranking entsprechend dargestellt werden, werden die einzelnen Elemente in der *ResultsList* sortiert und dabei doppelt vorhandenen URLs entfernt.

Da es während einer Suche passieren kann, daß u. U. mehrfach/parallel benutzte A– und T–Boxen sowie Operatoren- und Klassifikatorendatenbanken gespeichert werden müssen, wurde ein Lock–Mechanismus hinzugefügt. Dieser Mechanismus verhindert das Auftreten von inkonsistenten Zuständen.

Damit ein Benutzer einen Agenten komfortabler starten kann, wurde die Datei `servlet.properties` eingeführt. In dieser kann der Benutzer einige Informationen zum Starten des Agenten bekommen und eintragen. In dieser steht auch der Pfad zur lokalen Konfigurationsdatei des Agenten (siehe II D.1).

4.2.2 Wissensrepräsentation

Erstens haben wir festgestellt, daß wir mit unsicherem Wissen umgehen können müssen, da die Werte, die von den Klassifikatoren kommen und angeben, ob eine Instanz unter ein bestimmtes Konzept fällt, nicht entweder 0 oder 1 sind. Deshalb haben wir uns entschlossen, Instanzen mit Konfidenzwerten zu versehen, die den vom Klassifikator bestimmten Zugehörigkeitsgrad einer Instanz zum betreffenden Konzept angeben. Der Konfidenzwert ist ein *double* mit Werten aus dem Intervall $[0,1]$. Auch an den konkreten Rollen hängt jeweils ein Konfidenzwert aus $[0,1]$.

Bei den konkreten Rollen war diese Erweiterung kein Problem, weil jede konkrete Rolle nur genau einmal existiert. Bei den Konzepten war diese schon etwas schwieriger, weil die Konzeptliste, die zu jeder Instanz gehört, nur Zeiger auf Konzepte der Klasse `Concept` enthält. Aus diesem Grund wurde die Hilfsklasse `ConceptWithConfidence` eingeführt, die genau das darstellt, was der Name sagt.

Zweitens haben wir beschlossen, die Konzepte in "primitive" und "definierte" Konzepte zu verfeinern. Dabei gelten folgende Definitionen:

Definition: Ein Konzept K heißt **primitives Konzept**, wenn die von ihm repräsentierte Instanzenmenge eine Teilmenge des Schnitts aller durch seine Oberkonzepte $K_1, \ldots, K_n$ repräsentierten Instanzenmengen ist. Sei U die Instanzenmenge von K und $O_1, \ldots, O_n$ die Instanzenmengen von $K_1, ..., K_n$. Also $U \subseteq O_1 \cap \ldots \cap O_n$ gilt.

Definition: Ein Konzept heißt **definiertes Konzept**, wenn mit obigen Bezeichnungen $U = O_1 \cap ... \cap O_n$ gilt.

Im Gegensatz zu anderen Systemen mit Termsubsumtion, die definierte Konzepte über Rollen charakterisieren, können in unserem System auch primitive Konzepte Rollen haben. Der Vorteil eines definierten Konzeptes *c*, für das "*a* und *b* subsumieren *c*" gilt, ist, daß für eine Instanz *A*, die zu den Konzepten *a* und *b* gehört, ohne eine weitere Befragung von Klassifikatoren unmittelbar klar ist, daß die Instanz *A* auch zum Konzept *c* gehört. Das erspart uns in solchen Fällen, für das Konzept *c* einen eigenen Klassifikator anlegen und trainieren zu müssen.

Aus dieser Änderung folgt auch die Entscheidung, daß die Klassifikatoren von der T–Box aufgerufen werden. Ein Operator übergibt ein *NetResult* oder eine *Instance* an die Methode `TBox.classifiy`. Diese sorgt dafür, daß das *NetResult* bzw. die *Instance* klassifiziert und in die A–Box eingetragen wird. Dafür muß allerdings klar sein, in welche A–Box das *NetResult* eingetragen werden soll, bzw. aus welcher A–Box die *Instance* stammte.

4.2.3 Planer, Planausführung, Operatoren

Rückblickend muß die Planerkomponente im Vergleich zu den anderen Komponenten des Ausgangsmodells als teilweise recht „naive“ black–box charakterisiert werden. Ursache hierfür ist weniger mangelnde Sorgfalt bei der Modellierung, sondern vielmehr die Tatsache, daß die Aufgabenstellung (Planung von Suchen im Web) das eigenständige Weiterentwickeln von bestehenden Planungskonzepten – und damit Betreten von schwierigem Neuland – erzwungen hat. Konzeption und Implementierung verliefen daher weitgehend parallel.

Die folgenden Abschnitte beschreiben, welche konzeptionellen Änderungen bzw. Erweiterungen sich zu Abschnitt I 3.2.3 ff. ergeben haben.

Sensorgesteuerte Websuche?

Obwohl einzelne Bestandteile einer Websuche, wie etwa Klassifikatoren von Webseiten, gut als Sensoren eines Reiz-Reaktions-Schematas verstanden werden können, so blieb letzlich doch äußerst fraglich, wie eine *vollständige* Websuche in ein echtes Reiz-Reaktions-Schemata abgebildet werden kann. Ganz offensichtlich ist eine Websuche auch durch sehr aktive Prozesse wie dem gezielten Abfragen potentieller Informationsquellen charakterisiert.

Der anvisierte reaktive Planer „degenerierte“ deshalb zu einem, salopp gesagt, fast blindem Huhn: Der `PlannerTypeA` ist ein 1-Schritt Planer ohne Sensororientierung. Die Eigenschaft, Pläne der Tiefe 1 zu erstellen, teilt er mit echten sog. reaktiven Planern. Anders als diese, kann er seine Operatorwahl allerdings nicht von Sensoren abhängig machen – für die Suche im Web konnten keine sinnvollen Sensoren bestimmt werden. Der `PlannerTypeA` wählt Operatoren allein nach ihrer Bewertung aus. Zwei Eigenschaften verhindern, daß seine Planung völlig chaotisch verläuft. (1) Constraint Instanzen aus der Eingabe werden in die Operatorvorbedingungen eingesetzt, (2) nur solche Operatoren werden an die Planausführung weitergegeben, deren Vorbedingung einem „schwachen“ Erfüllbarkeitscheck standhält. Der „schwache“ Erfüllbarkeitstest erfasst – im Gegensatz zur starken Variante im Planausführer – keine prädikatübergreifenden Abhängigkeiten, diese können nur mit aufwendigeren rekursiven Verfahren geprüft werden.

Behandlung scheiternder Pläne

Echte klassische Planer blenden die Behandlung scheiternder Pläne aus. In ihrer statischen Modellwelt gibt es keinen Grund, warum ein Operator, dessen Vorbedingungen erfüllt sind, scheitern könnte. Dies gilt natürlich auch für *GraphPlan*, auf dem unsere Erweiterung *PG 343-GP* basiert. Da scheiternde Pläne bei Suchen im Web zum Normalfall gehören werden (z.B. ist eine Suchmaschine zeitweise nicht erreichbar oder liefert nicht die passende Seite, auf die der Operator gehofft hat,...), war es unumgänglich, scheiternde Pläne in unseren Planern systematisch zu behandeln.

Mindestens die folgenden beiden Methoden zur Behandlung scheiternder Pläne sind möglich:

1. Der Planer erzeugt statt einer Operatorsequenz einen Planbaum, der dann in kompakter Form die gewünschten Alternativpläne enthält.
2. Der Planer erzeugt nacheinander in quasi kanonischer Ordnung die gewünschten Pläne (also Sequenzen von Operatoren).

Der erste Ansatz konnte bisher nicht auf unseren C–Planer übertragen werden. Dies liegt zum einen daran, daß nicht offensichtlich ist, wie der zugrundeliegende *GraphPlan*-Algorithmus mit einfachen Modifikationen statt einer Operatorsequenz einen Operatorbaum erzeugen kann. Erschwerend kommt hierbei dann hinzu, daß man im allgemeinen wohl nicht den im worst case (in der Anzahl der Operatoren) exponentiell großen Operatorbaum, sondern nur Teile hiervon generieren will.

Den zweiten Ansatz konnten wir in die Planerkomponente integrieren. Die Lösung basiert auf zwei Erweiterungen im *PlanInformation*-Objekt. Dies ist zum einen eine `alreadyUsedOperators`-Datenstruktur, die von *PlanExecution*s angelegt und von den Planern ausgewertet wird. In dieser Datenstruktur wird quasi vermerkt, welcher Operator mit welchen Belegungen der Vorbedingungen und welcher anvisierten Nachbedingung schon mal zur Ausführung gekommen ist. Operatoren mit den genannten Eigenschaften wie sie in der Datenstruktur auftauchen, dürfen die Planer bei der Planung nicht mehr verwenden. Zum anderen gibt es nun ein Flag `worldHasChanged`, das den Planern anzeigt, ob sie ihr Weltmodell aktualisieren müssen. Eine *PlanExecution* setzt dieses Flag abhängig davon, ob sich bei einer (u.U. nur teilweisen) Planausführung das Wissen über Weltobjekte und ihre Beziehungen geändert hat. Einmal durch erfolgreiche Teilpläne angesammeltes Wissen geht also nicht verloren – nachfolgende Pläne müssen sich dieses Wissen nicht erneut erarbeiten, sondern können es direkt benutzen.

4.2.4 Datenbeschaffung

Während der Implementation stellte sich heraus – besser gesagt, es fiel uns wieder ein —, daß es für einen Agenten durchaus sinnvoll sein kann, sich einen Link auf einer Seite ersteinmal "genauer anzuschauen", bevor er die dahinter liegende Datei lädt. Für dieses "genauere Hinsehen" sollte die Klassifikationsfunktionalität von BotIshelly genutzt werden. Da bisher im Datenaustausch die Klasse `NetResult` den kleinsten gemeinsamen Nenner darstellte und es sich bei einem Link, der ursprünglichen Modellierung nach, nicht um eine Spezialisierung von `NetResult` handelte, gab es zwei Möglichkeiten:

1. Neben dem Datenaustausch auf `NetResult`–Ebene, wird eine neue Ebene für Links hinzugefügt. In allen am Datenaustausch beteiligten Klassen, wären dann neue, linkspezifische Methoden nötig geworden.

2. Ein gemeinsamer Nenner von Links und Net–Results wird als die Basisklasse des Datenaustausches definiert. Die bisherigen Methoden, die auf Net–Results arbeiten, müßten auf diesen Typ umgestellt werden.

Obwohl alle der Meinung waren, daß die zweite Alternative die "sauberere" ist, war noch einige Diskussion nötig, bis beschlossen wurde, sie auch zu verwirklichen. Es wurde die Klasse `NetEntity` als keinster Nenner eingeführt und die betroffenen Klassen entsprechend geändert. Damit ergab sich eine neue Hierarchie der am Datenaustausch beteiligten Klassen, die in einer Grafik dargestellt ist.

Neben dieser das Gesamtdesign von BotIshelly betreffenden Änderung, ergaben sich in der Datenbeschaffung noch weitere, kleinere Änderungen während der Implementierungsphase:

- Um das Trainieren von Klassifikatoren mit lokalen Daten zu ermöglichen, wurde ein Service, der Dateien als Net–Results von der Platte lädt, implementiert. Da die Basisklasse für derartige Services die Klasse `NetService` ist, bekam dieser Service den widersprüchlichen Namen `FileNetService`.

- Da inzwischen viele Web–Sites sog. "Frames" benutzen, die die gleichzeitige Darstellung mehrerer Dateien in einem Browser–Fenster ermöglichen, sollten diese auch durch BotIshelly unterstützt werden. Dies wird durch die Klasse `FrameSetNetResult` realisiert. Die BotIshelly–Net–Services benutzen diesen Typ.

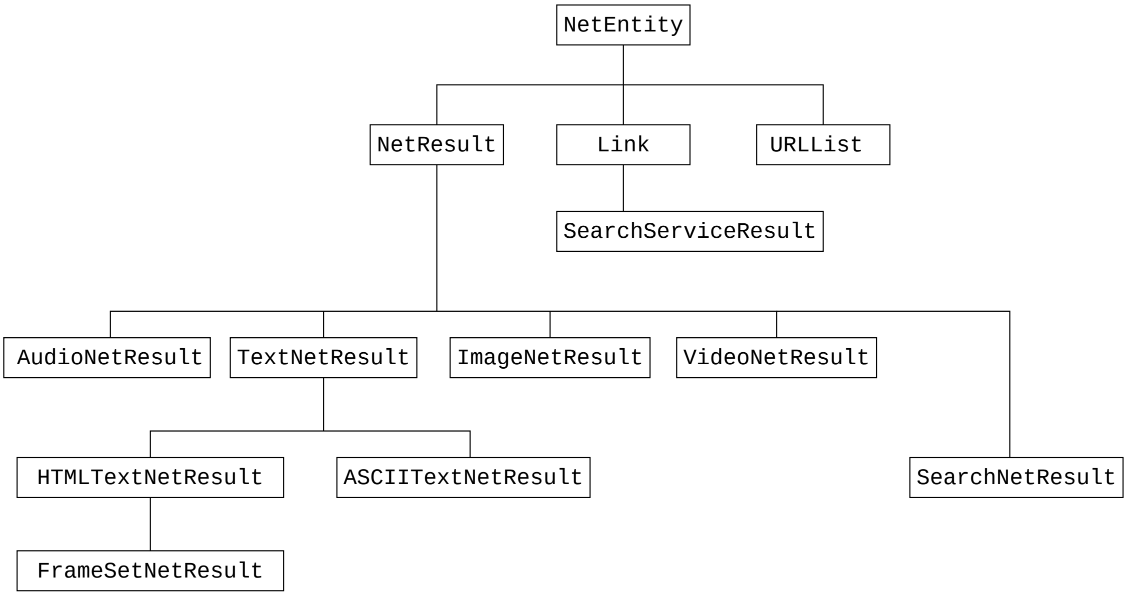


Abbildung 4.1: Klassen im Datenaustausch (nach Erweiterung der Modellierung)

- Da unverständlicherweise die Klasse `HttpURLConnetion` des JDK keine Timeouts zu unterstützen scheint[1] und wir das HTTP auch nicht neu implemtieren wollten, mußte diese Problematik, daß nämlich bei einer nicht antwortenden Verbindung u. U. beliebig lange gewartet wird, "unschön" gelöst werden. Dazu wird ein Thread gestartet, in dem versucht wird die Verbindung aufzubauen. Ist dies erfolgreich, so wird die Datei über diese Verbindung geladen. Hat die Verbindung nach einer gewissen Zeit jedoch (noch) nicht geantwortet, d. h. der Thread befindet sich immernoch in dem Aufruf zur Verbindungsherstellung[2], so wird ein Timeout ausgelöst und der Thread "gekillt" — und mit ihm der Versuch des Verbindungsaufbaus. Da ein Thread normalerweise kontrolliert terminieren soll, d. h. er wird dazu aufgefordert und beendet sich dann selbst[3], ist die entsprechende Methode `Thread.stop()` als "deprecated" gekennzeichnet, was beim Kompilieren eine Warnung erzeugt. Außerdem funktioniert diese Lösung zumindest unter Solaris 2 nicht mit dem JDK 1.2, sondern erst ab JDK 1.2.1.

  Sauber zu lösen ist das Problem mit einer HTTP–Implementierung, die Sockets mit Timeouts benutzt.

- Um die kumulierte Wartezeit auf Informationen aus dem Netz zu verkürzen, wurde die Möglichkeit hinzugefügt, mehrere HTTP–Verbindungen parallel zu nutzen. Es werden dafür Threads erzeugt, die je eine `HTTPNetService`–Instanz zum Laden benutzen, und die parallel ausgeführt werden.

  Dieses Vorgehen reduziert die Gesamtwartezeit des Agenten natürlich nur dann, wenn die Netzanbindung des "Agentenrechners" eine große Bandbreite besitzt und die angefragten "Datenserver" im Vergleich dazu relativ große Antwortzeiten und geringen Datendurchsatz besitzen.[4]

[1]oder wir diese Funktionalität nicht gefunden haben...
[2]ist also "I/O–blocked"
[3]funktioniert hier nicht, weil der Thread ja gar nicht läuft, sondern wegen I/O blockiert, und so auch nicht ansprechbar ist
[4]Für die Netzanbindung der Uni– bzw. PG–Rechner trifft dies z. B. zu.

4.2.5 Datenanalyse

Im Gegensatz zu anderen Teilbereichen von BotIshelly (wie z.B. dem Planer-Bereich) hat sich die Modellierung der Datenanalyse nur wenig geändert. Hauptsächlich wurden Erweiterungen der konkret realisierten Verfahren durchgeführt. Im Gegensatz dazu ist die Verfeinerung der Modellierung wohl ein natürlicher Prozeß in der Entwicklung, der in allen Bereichen notwendig war. Da diese Verfeinerung aber keine Änderung im eigentlichen Sinne des Modellierungskonzeptes ist, wird darauf nicht weiter eingegangen (die Aufzählung aller hinzugefügten Hilfsvariablen etc. würde ja doch nur langweilen, oder?). Die Änderungen und Erweiterungen werden in den folgenden Teilen erläutert.

`NetEntity` statt `NetResult`

In der Mitte der Implementierungsphase wurde die Schnittstelle aufgrund der Änderung der Vererbungshierarchie in der Datenbeschaffung geändert. Von diesem Zeitpunkt an wurden nicht mehr `NetResult`–Objekte übergeben, sondern `NetEntity`–Objekte. Dadurch war es möglich, alle möglichen Objekte der Datenbeschaffung klassifizieren zu können. In der Implementierungsphase des ersten Beispielagenten stellte sich nämlich heraus, daß es z.B. auch erforderlich war URL's zu „klassifizieren".

Neue Klassifizierertypen

Damit kommen wir zu einer Erweiterung der ursprünglichen Modellierung: die Klassifizierertypen `URLClassifier`, `LinkpageClassifier`, `NetEntityClassifier`, `KeywordClassifier` und `DummyClassifier`. Diese Klassifizierertypen wurden für verschiedene Zwecke benötigt.

Der `URLClassifier` klassifiziert nach der `URL`, d.h. sucht in der zu klassifizierenden `URL` die Teile, mit denen er trainiert wurde und klassifiziert entsprechend.

Der `LinkpageClassifier` klassifiziert eine `HTML`–Seite anhand den darauf vorhandenen `Link`s. Damit können z.B. Download–Seiten gut klassifiziert werden.

Der `NetEntityClassifier` klassifiziert anhand des Objekttyps in `Java`. Das war für die feine Granulierung der Operatoren im ersten Testagenten erforderlich. Mit diesem Klassifizierer können z.B. `FTP`–Daten klassifiziert werden, da sie einen anderen Typ in der `NetEntity`–Objekthierarchie besitzen.

Der `KeywordClassifier` klassifiziert eine Seite anhand der Schlüsselwörter, mit denen er trainiert wurde. Dies kann erforderlich sein, wenn ein breites Themenspektrum abgedeckt werden muß, wie es z.B. bei Produktseiten der Fall ist.

Der `DummyClassifier` klassifiziert immer mit dem vorgegebenen Wert. Dieser Klassifizierertyp ist zur Entwicklung weiterer Komponenten und Durchführung von Grobtests nützlich, da kein Training erforderlich ist.

Erweiterung der `Dictionary`–Hierarchie

Eine Änderung der Modellierung ergab sich bei der Entwicklung des FirmenSuchagenten. Aufgrund des Speichermangels wurde der Versuch gestartet, ein `Dictionary` zu entwickeln, das effizient mit dem Speicherplatz umgeht. Zu diesem Zweck wurde die Klasse `Dictionary` in eine abstrakte Klasse umgewandelt und jeweils eine erbende Klasse pro `Dictionary`–Typ hinzugefügt. Leider blieb dieser Versuch Speicherplatz einzusparen erfolglos, so daß als einzige Alternative die parallele Nutzung der Feature–`Dictionaries` realisiert wurde.

Die Hilfsmittel der Datenanalyse

Als weitere Erweiterung der Datenanalyse sind die Tools zu nennen. Während der Entwicklung der Agenten wurde es notwendig, Klassifikatoren zu erstellen und in einer Datenbank abzuspeichern. Zu diesem Zweck wurden mehrere Tools entwickelt, mit denen es möglich ist, Klassifikatoren zu trainieren, `ClassifierDatabase`s zu erzeugen und zu bearbeiten (`ClassifierGenerator` (s. II B.2),`dbExplorerConsole` (s. II B.1)) und Beispielmengen auf die lokalen Datenträger abzuspeichern (`GetLocalExamples` (s. II B.3)), um die Netzbelastung zu senken. Die Erzeugung von Klassifizierern wurde durch das `ClassifierConfigFile` (s. II D.3) vereinfacht.

4.2.6 Zusammenfassung

Zusammenfassend kann sicherlich gesagt werden, daß bei der ursprünglichen Modellierung das Thema Nebenläufigkeit, d. h. paralleles Ausführen von Programmteilen mit Hilfe von Threads, viel zu wenig beachtet wurde – bzw. gar nicht modelliert wurde. Code, der für sequentielles Ausführen designt wurde, „threadfest“ zu machen, stellte sich als durchaus trickreich heraus, da nicht alle Auswirkungen des parallelen Ausführens sofort erkannt wurden. Das Lösen dieser Problematik wurde auch nicht gerade dadurch unterstützt, daß verschiede JDK Version[5] sich beim Threading unterschiedlich verhalten.

Auffällig war außerdem, daß häufig die Modellierung zu grob war. Viele (Hilfs-) Klassen wurden hinzugefügt, so daß die Anzahl der Klassen nach der Implementierungsphase sich doch deutlich erhöht hatte. An der Grobstruktur, z. B. der Einteilung in Pakete, hat sich jedoch nicht allzuviel geändert.

Bei der Modellierung von Klassen und deren Interaktion steht der Aufwand, den deren Implementierung verursachen wird, natürlich im Hintergrund. Besonders die Gruppe, die sich mit den Planern zu beschäftigen hatte, mußte dies feststellen, wie oben beschrieben wurde. Natürlich ergaben sich auch bei den anderen Gruppen implemetationsbedingte Probleme, die gelöst werden mußten. Welche genau das waren, ist ebenso in den vorangegangenen Abschnitten beschrieben worden.

[5]hier JDK 1.2 und JDK 1.2.1 für Solaris 2

Kapitel 5

Implementierung des Firmen–Homepage–Agenten

5.1 Aufgabenstellung und Motivation

Gegenstand der Projektgruppe 343 war die Implementation einer Bibliothek, deren Komponenten die Implementation eines Agenten für die Suche im Internet, auch Softbot genannt, wesentlich vereinfachen sollten. Oft gebrauchte Funktionalitäten sollten als Grundbeisteine in Form von Javaklassen zur Verfügung gestellt werden. Ein Agent sollte durch das dem Zweck des Agenten entsprechende Zusammensetzen und Verbinden dieser Bausteine realisierbar werden.[1].

5.1.1 Aufgabenstellung

Nachdem diese Bibliothek, die inzwischen auf BotIshelly getauft worden war, fertiggestellt war, bestand die Aufgabe der Projektgruppe nun darin, an einem Beispiel zu demonstrieren, daß BotIshelly auch das Geforderte leisten kann. Die Projektgruppe beschloß, für diesem Zweck einen Agenten mit BotIshelly zu implementieren, der Homepages von Firmen im WWW findet, und zwar ohne das übliche "Durchklicken" von Unmengen Suchmaschinenergebnissen und das Ausprobieren von ad hoc gebildeten URLs. Genau diese immer gleichen Vorgehensweisen können nämlich automatisiert werden, was BotIshelly beispielsweise unterstützen sollte.

Das Hauptziel war die Demonstration der Fuktionalität der Bibliothek BotIshelly. Damit ist die korrekte Funktion der von BotIshelly bereitgestellten Einzelkomponenten, wie z. B. Planer, Klassifikatoren oder auch Datenbeschaffung, sowie das reibungslose Zusammenspiel der BotIshelly–Komponenten gemeint, das für die Gesamtfunktion eines Agenten notwendig ist.

5.1.2 Motivation

Die Suchdomäne wurde gewählt, weil sie als Suchgebiet eine wichtige Aufgabe darstellt, wie auch eine in diese Richtung gehende Anfrage aus der Wirtschaft an die Projektgruppe 343 bestätigte. Ein weiterer Punkt war die Möglichkeit, daß sich hier der Suchraum in günstiger Weise eingrenzen läßt. Denn schon beim Testen der Shell stellte sich heraus, daß der Suchraum i. a. sehr groß werden kann. Eine zielgerichtete Suche ist dann nicht mehr effizient durchführbar. Da es aber weder wünschenswert noch wirklich möglich ist, das Web an sich zu beschränken, muß die Beschränkung

[1] Soll aus einer Menge Steine ein komplexes Bauwerk werden, ist immernoch ein guter Architekt, ein Maurer und eine Menge Mörtel nötig...

der Suche im Themengebiet liegen. Dies erfordert z.B. eine sehr sorgfältige Modellierung der T–Box in der Wissenrepräsentation (s. I 3.2.2). Da diese als der terminologische Teil, in der die Begriffe definiert und in eine Begriffsstruktur eingeordnet werden, das Domänenwissen des Suchagenten darstellt, hängt von ihrer Modellierung die Güte des Agenten zu einem wesentlichen Teil ab. Ist die Modellierung zu komplex, wächst die Größe des Suchraums, wie schon oben erwähnt, zu stark. Wird andererseits die T–Box zu einfach gehalten, ist der Agent nicht in der Lage, während der Suche potentiell mögliches Wissen zu erwerben.

Auch die zu programmierenden Operatoren (s. I 3.2.3) wirken sich auf die Effizienz der Planer unmittelbar aus. Je nachdem, was für Operatoren zur Verfügung stehen, kann die Zahl der erstellbaren Pläne explodieren, die Planner können systematisch unsinnige Pläne erzeugen oder aber es können keine Pläne erstellt werden, die zum Ziel führen.

Bei der gewählten Suchdomäne kann in der Wissenrepräsentation die Zahl der Konzepte, welche sich auch u. a. direkt auf die Zahl der zu programmierenden Operatoren auswirkt, klein gehalten werden, ohne die Möglichkeiten des Agenten zu stark einzuschränken. Dieses waren aufgrund der wenigen Zeit, die zur Programmierung des Agenten verblieb (i. W. 64 Personenarbeitstage), mit ausschlaggebende Gründe zur Auswahl der Such–Domäne.

5.2 Realisierung

Bei der Realisierung des Agenten mußte zunächst die Domänen genauer eingeschränkt werden. So wurde die Anzahl der Branchen auf zwanzig[1] beschränkt, um den Aufwand zum Erstellen der Beispielmengen für die Textklassifikatoren nicht zu groß werden zu lassen. Um die Erstellung weiter zu vereinfachen, wurden außerdem die Branchenbezeichnungen von Yahoo übernommen und von den dort aufgeführten Seiten jeweils 100 pro Branche als Traingsmenge verwendet.

5.2.1 Wissenrepräsentationskomponente

Eine am Anfang der Realisierung des Agenten notwendige Aufgabe ist die sorgfältige Modellierung der T–Box. Da darauf alle weiteren Arbeiten aufbauen, insbesondere auch die noch zu programierenden Operatoren, wäre eine fehlerhafte Modellierung im nachhinein nur mit sehr viel Aufwand zu korrigieren. Auch der Projektgruppe gelang es nicht auf Anhieb, sondern erst nach mehreren Durchläufen und intensiven Diskussionen, eine relativ robuste T–Box zu erstellen (s. Abbildung I 5.1). Dabei war es auch sehr hilfreich, eine graphische Darstellung zu erstellen, weil so am schnellsten deutlich wird, wo Modellierungsfehler sind, z. B. wenn ein Konzept keinerlei *isA*–Beziehungen hat, nicht einmal zu `anything`, oder wenn es durch die modellierten Rollen möglich wird, unsinnige konkrete Rollen zu den Instanzen zu bilden, was nicht erwünscht ist.

Außerdem wurde eine rudimentäre A–Box[2] erstellt, um dem Agenten einige notwendige Instanzen zur Verfügung zu stellen, wie Instanzen zu den Branchen (Konzept *trade*) und Instanzen für Schlüsselwörter (Konzept *keyword*), um den Suchbegriff einzugrenzen. Diese Schlüsselwörter sind durch konkrete Rollen vom Typ *relates_to* mit der jeweiligen Branchen–Instanz verbunden.

5.2.2 Operatoren

Die hauptsächliche Arbeit bei der Realisierung bestand dann darin, problemspezifische Operatoren zu implementieren. Vorher waren jedoch die Signaturen und damit deren jeweilige Funktionalität

[1] Antiquitäten + Sammeln, Bauwesen, Computer, Drucken, Essen + Trinken, Fotografie, Geschenke, Information, Konferenzen + Messen, Landwirtschaft, Marketing, Nachrichten + Medien, Outdoor, Politik + Verwaltung, Reisen, Spielzeug, Telekommunikation, Umwelt, Verpackungen und Wissenschaft

[2] Die Evaluation wurde nur mit dieser rudimentären A–Box durchgeführt.

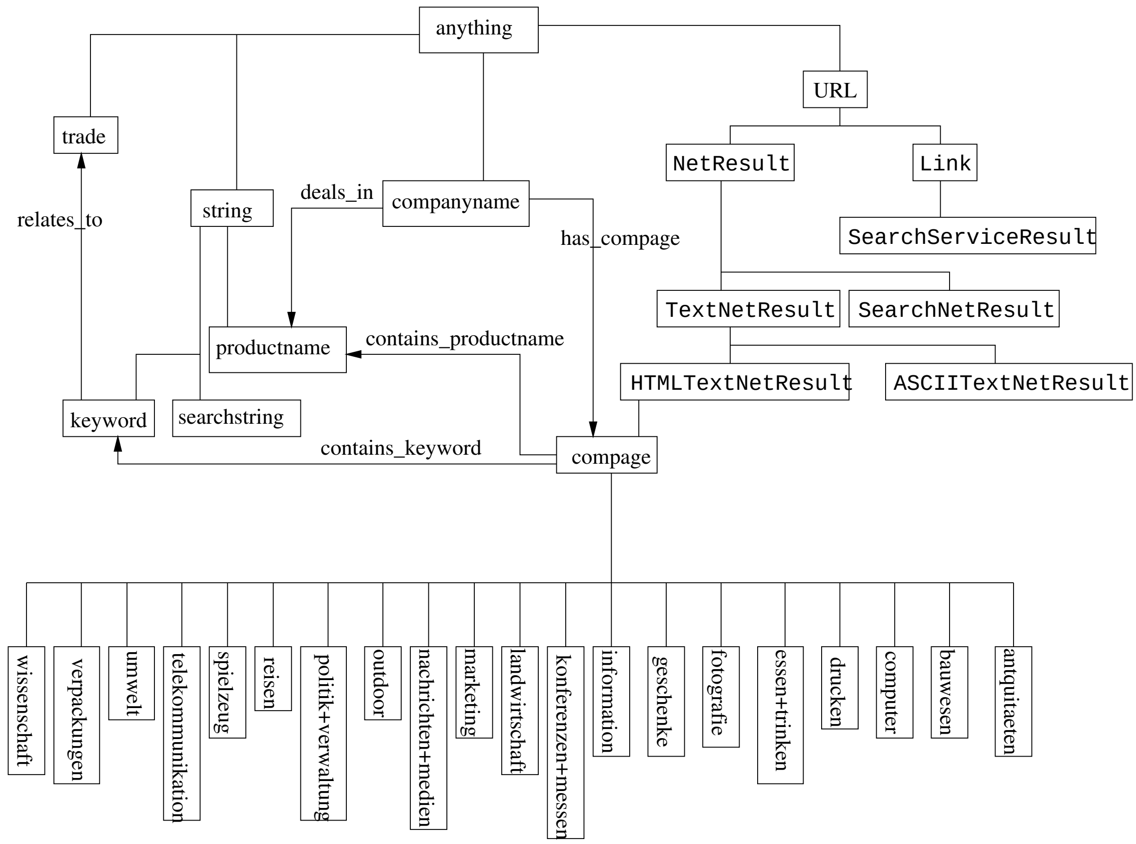


Abbildung 5.1: Die T–Box des Firmenagenten

festzulegen. Natürlich benötigt man z. B. Operatoren, die bestimmte Suchmaschinen mit entsprechenden Suchstrings abfragen. Bei diesem Agenten werden die drei Suchmaschinen Nathan[3], Yahoo[4] und AltaVista[5] angefragt. Außerdem gibt es "Rate"–Operatoren, die aus dem Firmennamen durch einfaches Zusammensetzen mit ".com",".de" etc. die mögliche URL der Firmen–Homepage raten. In vielen Fällen kann damit schnell die richtige Seite gefunden werden. Desweiteren gibt es auch einen Operator, der aus der URL einer HTML–Seite, die für eine Seite über ein bestimmtes Produkt gehalten wird, eine neue URL bildet, für die er dann testet, ob es sich um eine Firmenhomepage handelt. Dies ist nur ein kleiner Ausschnitt aus der Menge der implementierten Operatoren.

Da sowohl die Länge der Pläne als auch die Anzahl der möglichen Pläne, die der Agent zur Suche abarbeitet, von den Operatorensignaturen bestimmt werden, ist hier sehr viel Abstimmungsarbeit zu leisten. Die Performanz des Agenten steht und fällt mit der Qualität und Kombination der Operatoren, für die sich entschieden wird. Bei der Entwicklung dieses Agenten wurde die anfänglich angedachte Menge an relativ einfachen Operatoren durch eine kleinere Menge komplexerer Operatoren ersetzt, da sonst die Anzahl der möglichen Pläne "auszuufern" drohte. Das Potential des Agenten ist an dierser Stelle sicher nicht (voll) ausgeschöpft worden. So könnte man sich vorstellen, daß mit Hilfe eines Operatoren–Lerners, der leider aufgrund der zeitlichen Beschränkung nicht verwirklicht werden konnte, und eventuell einiger von Hand durchgeführten Beispielsuchen, die Operatoren so gewichtet und zu „Compound"–Operatoren zusammengesetz würden, daß unsinnige Pläne kaum noch erstellt werden könnten.

5.2.3 Input/Output

Da BotIshelly komplett in Java geschrieben ist, bot es sich natürlich an, die Java–Servlets als Schnittstelle zu verwenden, zumal BotIshelly diese Lösung unterstützt. So können auch mehrere Suchanfragen parallel bearbeitet werden.

Die Eingabemaske des Agenten wurde so gestaltet, daß auch unterschiedliche Anfragekombinationen gestellt werden können, wobei aber nur ein begrenzte Zahl an Kombinationen zulässig ist. Neben dem Firmen- und Produktnamen, sowie der Branchenauswahl kann noch zusätzlich ein Stichwort angegeben werden.

5.2.4 Ranking

Die Ergebnisse werden in Form ihrer URL–Adresse und dem Textanfang der Seite selber angezeigt. Die Reihenfolge wird durch ein Ranking auf Prädikatenebene festgelegt. Das Ranking kann im Prinzip beliebig gestaltet werden. Bei diesem Agenten wurde das arithmethische Mittel über die Konfidenzwerte der Zielprädikate gewählt:

$$rank = \frac{\sum_{p \in TPreds} conf(p)}{|TPreds|}$$

wobei

- $conf(p)$ der Konfidenzwert des Prädikats p und
- $TPreds \subseteq \{\ has\_compage(companyname, compage),\ contains\_keyword(compage, keyword),$ $contains\_productname(prodpage, prodname),\ has\_productpage(companyname, prodpage)\ \}$ die (von der Eingabe abhängige) Menge der Zielprädikate ist.

Dieses ist nur ein einfacher Ansatz, und kann sicherlich noch erheblich verfeinert werden.

[3] http://www.nathan.de
[4] http://de.yahoo.com/
[5] http://www.altavista.de/

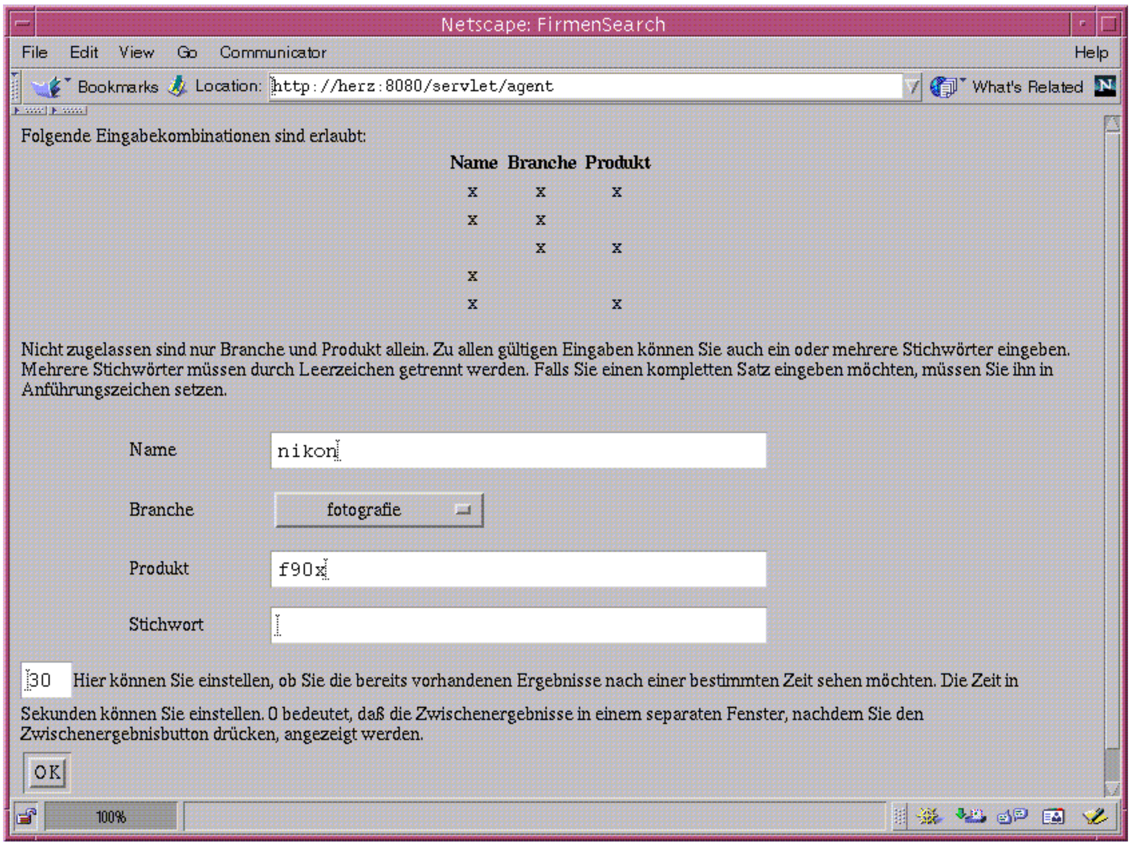


Abbildung 5.2: Die Eingabe–Maske des Firmen–Agenten

Abbildung 5.3: Die Ergebnis–Maske des Firmen–Agenten

Kapitel 6

Analyse der PG–Ergebnisse

6.1 Evaluation der Shell

Die Hauptaufmerksamkeit bei der Bewertung der Ergebnisse der PG sollte auf die Shell gerichtet sein, weil die Aufgabe in erster Linie darin bestand, die Shell zu entwickeln. Die Anforderungen an die Shell waren:

1. *Modularität*
   Dies ist insofern erfüllt, als daß man bei Änderungen eines Moduls, falls man nicht gleichzeitig die Schnittstellen verändert, die restlichen Module nicht anpassen muß. Zum Beispiel könnte eine andere Eingabemaske benutzt werden, wofür nur die Schnittstelle in der Systemkontrolle angepaßt werden muß. Noch einfacher ist es, wenn andere Klassifizierer oder ein neuer Planertyp hinzugefügt werden sollen. Hierbei müssen bei der Implementierung nur die in der Dokumentation beschriebenen Schnittstellen eingehalten werden.

2. *Domänenunabhängigkeit*
   Die Domänenunabhängigkeit ist dadurch sichergestellt, daß die Wissensrepräsentationskomponente (WRK) alle notwendigen Strukturen und Methoden zur Verfügung stellt, um Spezialistenwissen für konkrete Anwendungen aufzunehmen und anschließend manipulieren zu können. Außerdem gibt es bei den Search–Services die Möglichkeit, neue Schnittstellen zu anderen Suchmaschinen hinzuzufügen bzw. vorhandene zu ersetzen.

3. *Anpassungsfähigkeit an ein sich schnell veränderndes Web*
   Erstens wird die Suchfunktion nicht durch Anfragen an einen vorgefertigten Katalog, der ständig aktualisiert werden müßte, realisiert, sondern es wird tatsächlich im Internet gesucht. Zweitens ist die Anpassung an Veränderungen im Web durch die in Punkt 2 beschriebenen Komponenten möglich. Allerdings müssen die Schnittstellen für neue Suchmaschinen vom Shellbenutzer selbst implementiert und eingefügt werden. Falls das konzeptionelle Wissen in der WRK geändert werden soll, muß das auch der Shellbenutzer machen und für evtl. neue Konzepte entsprechende Klassifikatoren erzeugen und trainieren. Sollten die Änderungen auch die Operatoren betreffen, müßten auch die entsprechend angepaßt werden, bzw. neue implementiert werden. Das heißt, daß die Anpassungfähigkeit nur in eingeschränktem Maße gegeben ist. Sie ist zwar möglich, aber mit einiger Arbeit für den Shellbenutzer verbunden.

4. *Verwirklichung intelligenter Suchstrategien*
   Für die Verwirklichung intelligenter Suchstrategien sind die Planer zusammen mit den Operatoren zuständig. Die Planungskomponente bildet intelligente Suchstrategien nach, wobei der Planer garantiert, daß ein Plan erstellt wird, der das Ziel erreicht, falls das mit den vorhandenen Operatoren überhaupt möglich ist. Die „Intelligenz“ steckt in den Operatoren,

die vom Shellbenutzer geschrieben werden müssen, weil sie spezifisch für die Suchdomäne sind.

5. *Inhaltliche Bewertung der Suchergebnisse*
Die Klassifikatoren nehmen die inhaltliche Bewertung der Suchergebnisse vor, indem sie diese z. B. mit Trainingsbeispielen vergleichen und als Ergebnis einen Konfidenzwert liefern. Die Konfidenzwerte sind ausschlaggebend dafür, welche Ergebnisse in welcher Reihenfolge präsentiert werden.

6. *Bereitstellung aller Grundbausteine, um konkrete Agenten mit wenig Aufwand zu realisieren.*
Um diese Anforderung überprüfen zu können, haben wir aus der Shell heraus einen Prototyp eines Firmen–Homepage–Finder erstellt. Es ist uns gelungen innerhalb von 64 Personenarbeitsstunden den Agenten zusammenzubauen. Dabei hat das Schreiben der Operatoren die meiste Arbeit verursacht. Danach war der Firmen–Homepage–Agent einsatzbereit. Das wäre ohne die Shell in dieser kurzen Zeit nicht möglich gewesen.

7. *Funktionsfähigkeit der Shellbestandteile und reibungslose Interaktion zwischen ihnen.*
Das ist durch die Funktionsfähigkeit des Firmen–Agenten bewiesen.

6.1.1 Firmen–Agent vs. MetaGer

Um die Qualität unseres Firmen–Homepage–Agenten beurteilen zu können, haben wir folgendes Testszenario gewählt: Wir, die Mitglieder der PG, simulieren die Benutzer des Agenten und vergleichen die Qualität der Suchergebnisse mit den Ergebnissen der Meta–Suchmaschine MetaGer (`http://www.metager.de`). Dabei geben wir eine vorher festgelegte Testdatenmenge sowohl in den Agenten als auch in *MetaGer* ein. Die Testdatenmenge setzt sich aus folgenden Daten zusammen:

Best–Case Anfragen Bei den Firmennamen führt das Anhängen von "`.de`" oder "`.com`" zum Erfolg.

Average–Case Anfragen Die betreffende Firma besitzt eine Homepage, deren URL nicht so einfach geraten werden kann.

Worst–Case Anfragen Bestehen aus Firmen, die nicht im Internet vertreten sind, oder aus unsinnigen Eingaben (z. B. Produkte, die von den Firmen nicht hergestellt oder gehandelt werden).

Im einzelnen sind es folgende Testdaten:
Best-Case-Eingaben:

Name	**Branche**	**Produkt**	**Stichwort**
Antik-Conrady	Antiquitäten	Tische	—–
Hochtief	Bauwesen	Software	—–
Eiffel	Computer	Compiler	—–
Maxdata	Computer	Notebook	—–
Epson	Drucken	Tintenstrahldrucker	—–
—–	Essen + Trinken	Milchshake	—–
Nikon	Fotografie	Spiegelreflexkamera	—–
Vedes	Geschenke	—–	Spielzeugladen
IAA	Konferenzen + Messen	—–	Auto
Fendt	Landwirtschaft	Traktor	—–
FAZ	Nachrichten + Medien	Tageszeitung	—–
ZDF	Nachrichten + Medien	heute	—–

Rollerblade	Outdoor	Inlineskates	---
SPD	Politik + Verwaltung	Steuerreform	---
Mattel	Spielzeug	Barby	---
Telekom	Telekommunikation	D1	---
Alphatelecom	Telekommunikation	---	Telefongesellschaft
Greenpeace	Umwelt	Demo	---
Tetrapak	Verpackung	Milchtüte	---
Enerko	Politik + Verwaltung	---	Acropolis
Siemens	Telekommunikation	IC35	---

Average-Case-Eingaben:

Name	**Branche**	**Produkt**	**Stichwort**
Beos	Computer	Stinger	---
Ameling	Essen-Trinken	---	---
---	Geschenke	Parfumzerstäuber	---
Niedermeyer	Fotografie	---	---
Panorama Vision	Fotografie	---	---
I-Tüpfelchen	Geschenke	---	---
Classic Link	Marketing	Voice Response System	---
Wundermann Cato Johnson	Marketing	---	---
Skyship	Reisen	---	---
Nicolaus	Verpackungen	Faltschachtel	---
Thermotemp	Wissenschaften	---	---
Pieter van Weenen	Information	---	---
All.ex	Landwirtschaft	Taubenabwehr	---
Athener Zeitung	Nachrichten-Medien	Zeitung	---

Worst-Case-Eingaben:

Name	**Branche**	**Produkt**	**Stichwort**
Antik-Daniel	Antiquitäten	Möbel	---
Interkrenn	Bauwesen	Bohrmaschine	---
SAP	Computer	R/4	---
Trioptimum	Computer	---	Hardware
Rocketlauncher	Computer	---	Microsoft
Druckerei Zellerhoff	Drucken	---	---
Aldi	Essen + Trinken	Nutoka	---
Foto-Ullrich	Fotografie	Objektiv	---
Vatikan	Messe + Konferenzen	---	Heilige Messe
Essing	Outdoor	Fahrrad	---
Nokia	Telekommunikation	1234	---

Wir haben uns, um die Qualität der Suchergebnisse zu messen, für folgende Kriterien entschieden: Für die *Best–Case*- und die *Average–Case*–Daten

- Es werden nur die ersten zehn URLs betrachtet. Die Positionen von 1 bis 10 werden in umgekehrter Reihenfolge durchnumeriert.
- Die einzelnen URLs werden mit „ist die gewünschte Seite“(Wert 4), „enthält richtigen Link“(Wert 2), „enthält Informationen zum Thema und hilft vielleicht weiter“(Wert 1) und „paßt nicht“(Wert 0) bewertet.
- Um die Qualität des Rankings in der Gesamtbewertung zu berücksichtigen, wird die Bewertung der URLs mit dem Wert der Position multipliziert.
- Das Qualitätsmaß berechnet sich dann wie folgt:

$$\frac{\sum Wert \cdot Position}{4 \cdot |URLs|}$$

Für die *Worst–Case*–Daten

- Es wird nur die Anzahl der gefundenen URLs gezählt und als negativer Wert interpretiert.

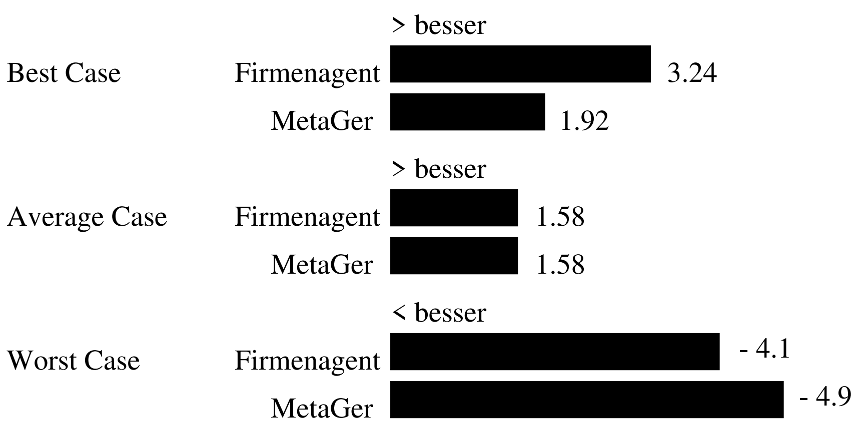


Abbildung 6.1: Ergebnisse der Tests

Bewertung der Ergebnisse und Begründung

Abbildung I 6.1 zeigt die jeweiligen Mittelwerte der Bewertungen der Suchergebnisse. Der Grund für die guten Ergebnisse des Firmen–Agenten bei den Best–Case–Beispielen, sind in erster Linie die heuristischen Operatoren, die zu einem gegebenen Firmennamen durch Anhängen von “`.de`”, bzw “`.com`” die Adresse der Homepage raten. Diese Strategie ist sehr erfolgreich, da sich viele Firmen–Homepages auf diesem Weg finden lassen. Auch MetaGer verwendet dieses Vorgehen bei seinen Quicktips. MetaGer überprüft aber die Adressen nicht. Dagegen bewerten die Klassifikatoren des Firmen–Agenten die so gefundenen Seiten. Deshalb wurden die Quicktips von MetaGer nicht in die Bewertung der Suchergebnisse miteinbezogen.

Für die Average–Case–Beispiele ergibt sich im Mittelwert über alle Suchen ein Gleichstand. Sieht man sich die einzelnen Suchergebnisse an (s. I B), stellt man fest, daß der Firmen–Agent zum Teil die gesuchte Seite lieferte, in zwei Fällen sogar an erster Stelle, aber dadurch, daß er noch weitere Ergebnisse liefert, die in keinem Zusammenhang zur Suchanfrage stehen, hat sich die Bewertung deutlich verschlechtert. Der Grund für das Anzeigen von nicht passenden Links als Suchergebnisse liegt zum einen natürlich an der Ranking–Formel (s. I 5.2.4), die letztlich für die Auswahl der

Suchergebnisse verantwortlich ist, aber zum anderen auch an den Beispielmengen, mit denen die Klassifikatoren trainiert wurden.

Ähnliches gilt für die Worst–Case–Beispiele. Um dem Firmen–Agenten in schwierigen Fällen weitere Anhaltspunkte an die Hand zu geben, wurden zu jeder Branche Stichworte, nach denen gesucht werden kann, in die Wissensrepräsentation aufgenommen. Alle so gefundenen Seiten werden als Ergebnisse angezeigt, auch wenn sie mit der Suchanfrage nichts zu tun haben.

6.2 Stärken und Schwächen von BotIshelly

Die Stärken und Schwächen von BotIshelly lassen sich in zwei Kategorien unterteilen

1. programmiersprachenbedingte Stärken und Schwächen und
2. modellierungsbedingte Stärken und Schwächen.

6.2.1 Programmiersprachenbedingte Stärken und Schwächen

Java ist eine plattformübergreifende Programmiersprache mit starker Ausrichtung auf den Einsatz in einem TCP/IP–basierten Netzwerk. Durch diese Tatsache ist BotIshelly auf allen Plattformen, für die es eine Java Virtual Machine (JVM) gibt, lauffähig. Dadurch ist BotIshelly durch einen großen Personenkreis nutzbar.

Leider unterstützt die mit Java mitgelieferte HTTP-Implementierung keine netzbedingten Timeouts, so daß es unter Benutzung dieser Standardmethoden passieren kann, daß der Agent ewig wartet, wenn eine Seite nicht geladen werden kann. Jens Jägersküpper hat dieses Manko durch eigene Methoden umgangen, die einen Timeout durch Threads realisieren.

Die Unterstützung von Multithreading ermöglicht es BotIshelly grundsätzlich multiuserfähig zu sein und mehrere Planer und Planausführer zu beschäftigen. Leider sind die von Sun implementierten Thread-Methoden zum Anhalten und Zerstören von Threads mit Java 2 für obsolet erklärt worden, da sie deadlockgefährdet sind. Dadurch ist es notwendig geworden, daß `Planner` und `PlanExecution` eigene Thread-Methoden implementieren. Noch gravierender ist dieses Problem bei den zuvor angesprochenen Timeouts. Es ist zwar feststellbar, daß das Holen einer Seite eigentlich einen Timeout erzeugt, aber dieser Thread ist nur unter Einsatz einer obsoleten Methode zerstörbar, d. h. in kommenden Java–Versionen ist diese Problemlösung evtl. nicht mehr möglich. Alternativ bleibt der Thread erhalten und belegt weiterhin Speicher.

Das Speicherproblem ist ein weiteres Manko der gewählten Programmiersprache. BotIshelly ist – wie für Java–Anwendungen üblich – speicherhungrig. Zusammen mit dem Anwendungsfeld künstliche Intelligenz ergeben sich daraus erhebliche Ressourcenanforderungen für einen echten Multi–User–Betrieb. Auf den am Fachbereich momentan installierten PG–Pool–Rechnern Sparc–Ultra10 mit 128MB Hauptspeicher ist es daher nicht sinnvoll, mehr als eine Suche gleichzeitig auf einem Rechner laufen zu lassen. Der Versuch mehrere Suchen gleichzeitig auf einem Rechner auszuführen endet i.d.R. in einer Out-of-Memory-Exception. Bei Bereitstellung von mehr als 64MB Heapsize für die JVM wird die Suchdauer durch Swapping beträchtlich erhöht.

6.2.2 Modellierungsbedingte Stärken und Schwächen

Im Verlauf der Implementierung ist aufgefallen, daß es sinnvoller gewesen wäre, Oberklassen für `Concept` und `Role` sowie für `Instance` und `ConcreteRole` einzuführen, da über diese Paare oft iteriert wird und im Moment durch die fehlende Oberklasse Fallunterscheidungen nötig sind.

Es fällt auf, daß die Wissensrepräsentation innerhalb von BOTISHELLY in zwei Ausprägungen vorhanden ist. Einmal wird Wissen in Form von Prädikaten repräsentiert und ein zweites Mal in einer KL–One–ähnlichen Form. Es stellt sich die Frage, ob diese Redundanz sinnvoll ist. Andererseits unterstützt diese Redundanz Personen, die sich mehr in dem einen oder dem anderen Bereich der künstlichen Intelligenz auskennen. Die Repräsentationsformen haben auch in BOTISHELLY einige Vorteile. Die Prädikatrepräsentation unterstützt z.B. keine Subsumption. Die KL–One–artige Repräsentation kann nicht mit unvollständig belegten Rollen bzw. Prädikaten umgehen. Dieses „Halbwissen" ist wichtig für den Planer und den Planausführer. Beide Formen der Wissensrepräsentationen sind also aufgrund der unterschiedlichen Fähigkeiten durchaus sinnvoll und wünschenswert, auch wenn durch die Verwendung beider Repräsentationsformen oft zwischen beiden konvertiert werden muß. Zudem ermöglicht die Trennung der Wissensrepräsentation in der Wissensrepräsentationskomponente und in den Operatoren den Austausch einer der beiden Module ohne den nicht ausgetauschten Teil nachhaltig zu beeinträchtigen.

Der von der PG entwickelte Planer Typ C, der Graphplan um die die Möglichkeit der Verarbeitung dynamischer Objekte erweitert und trotzdem kürzeste Pläne generiert, ist eine weitere Stärke der Shell. Dieser Planer ist außerdem in der Lage weitere Pläne zu generieren, die über einen Umweg zum Ziel kommen, sollte der kürzeste Plan scheitern. Leider kann der Planer nicht mit „atomaren" Operatoren umgehen. Es ist nicht möglich einen Operator zu schreiben, der eine URL erzeugt und einen weiteren Operator, der eine beliebige URL in der Vorbedingung stehen hat und in der Nachbedingung eine klassifizierte URL. Die durch solche Operatoren erhöhte Gesamtanzahl und Kombinationsmöglichkeiten von Operatoren bewältigt der Planer nicht. Es kommt zu Speicherproblemen und die Generierung eines Plans erfordert erheblich mehr Zeit. Da der Planer weder über schon eingesetzte Operator–Prädikat–kombinationen buchführt und dadurch feststellt, ob ein nun neu erzeugtes dynamisches Objekt vermutlich dasselbe sein wird wie ein zuvor mit derselben Kombination erzeugtes, noch die gekennzeichneten Prädikate, die aus der Benutzereingabe generiert werden besonders beachtet, explodiert die Komplexität des Planes und verursacht Speicher- und Laufzeitprobleme. Sind die Operatoren jedoch so geschrieben, daß der Planer komplexitätstechnisch mit ihnen zurecht kommt, so werden die Pläne sehr schnell generiert.

Der Planer Typ A, der schon in der Entwurfsphase als eher „dummer" Planer charakterisiert wurde, leidet unter dem Fehlen eines Operatorenlerners (s. II 15), da er so keine Möglichkeit hat seine Pläne zu verbessern. Trotzdem liefert auch dieser Planer zusammen mit der Shell erstaunlich schnell Ergebnisse.

Der Planausführer kann vermutlich mit den meisten Planern zusammenarbeiten, so daß er nur selten geändert werden muß. Er kann jedoch nicht mit mengenwertigen mehrstelligen Prädikaten umgehen. Das ist schade, da z.B. bei Suchmaschinenanfragen häufig mehr als eine URL zurückgeliefert wird, die nun nicht in die A–Box übernommen werden. Der Grund für dieses Manko ist, daß die PG noch keine Daten- und Logikkonsistente Lösung für das Problem gefunden hat. Um das Problem zu verdeutlichen, folgt ein Beispiel:

OP_1 Summenerzeuger zahl2+zahl3=10

Vorbedingung zahl(x::zahl), zahl(y::zahl)

Nachbedingung Summe10(x::zahl,y::zahl)

OP_2 Ganzzahlig teilbar durch 2

Vorbedingung zahl(x::zahl)

Nachbedingung teilbar(x::zahl,2)

$x := \{1, 2\}$

$y := \{9, 8\}$

Lösungsansatz	Problem
summe10({1,9};{2,8})	Paarweise Zuordnung funktioniert nur bei gleicher Mächtigkeit der Mengen, die dann als Vektoren aufgefasst werden.
summe10({1,9};{2,8};{1,8};{2,9})	Hier ist das Problem der nicht gleich mächtigen Mengen durch Kombination von jedem Element der einen Mengen mit der jedem der anderen Menge gelöst, allerdings sind dabei dann auch Falschaussagen möglich.
Operator nutzt nur genau eine Belegung summe10(1,9) und setzt die benutzten Mengenelemente je Menge auf eine spezielle Position.	Der Planer verzeigert die Prädikatsargumente so, daß sich eine Änderung in einer Menge an allen Stellen auswirkt. Das ist für die Vorwärtspropagation von Belegungen auch genauso gewünscht. Natürlich ändert sich das dann aber auch rückwärts, wenn ein nachgeschalteter Operator die Menge ändert. Wenn nach OP_1 OP_2 ausgeführt wird, dann wird dieser Operator die 2 aus der Menge x auf die spezielle Position rücken, dann wird aber rückwirkend folgende Behauptung aufgestellt Summe10(2,9), was wieder falsch ist.
In den `Predicate` stehen die tatsächlichen Belegungen statt in den `PredArguments`.	Die Propagation der Prädikatsbelegungen nach vorne ist unmittelbar abhängig von der jetzigen Implementierung und noch viel wichtiger als eine Lösung, die im Planausführer Mengen ermöglicht.

Tabelle 6.4: Lösungsansätze für das Mengenproblem im Planausführer

Tabelle I 6.4 zeigt ein paar Lösungsansätze auf, die leider alle wieder aus den dort genannten Gründen verworfen werden mußten.

Die von einem Agenten gefundenen Lösungen werden i.a. in der A–Box zwischengespeichert werden, um dann später ausgewertet zu werden. Diese Zwischenspeicherung und die Verwendung von Threads zur Steuerung von Planer und Planausführer erlauben es, zu jeder Zeit die Auswertung der schon gefundenen Ergebnisse vorzunehmen. Damit unterstützt die Shell Anytime-Algorithmen. Der Benutzer kann Zwischenergebnisse abfragen ohne die Suche unterbrechen zu müssen.

BotIshelly stellt eine umfassende und schnell erweiterbare Logging-Umgebung zur Verfügung. Über eine statische Klasse kann jede neue Klasse seine Logausgaben in das agentenweite Logfile einfließen lassen. Leider ist es dadurch nicht möglich die Logausgaben einer bestimmten Suche zuzuordnen, weil alle Suchen in dasselbe Logfile schreiben, da der Pfad zum Logfile statisch global gesetzt wird.

BotIshelly stellt die folgende Bereiche der künstlichen Intelligenz und des Information Retrievals in einer einzigen Klassenbibliothek zur Verfügung:

1. (Text-) Klassifikation
2. Lernen
3. Planen
4. Wissensrepräsentation
5. Ranking
6. Informationretrieval aus externen Datenbanken

Durch den modularen Aufbau von BotIshelly ist es möglich, innerhalb kurzer Zeit funktionsfähige Agenten für eine beliebige Domäne zu erstellen. Außerdem ist es möglich einzelne Module auszustauschen, um sie zu verbessern, zu ergänzen oder zu testen. Der Arbeitsaufwand für den Austausch der Module ist nicht für alle Module gleich hoch. So sind z.B. die Operatoren und Klassifikatoren sehr einfach auszutauschen, wohingegen die Wissensrepräsentationskomponente, aufgrund der an sie gestellten Anforderungen schwierig auszutauschen ist. Trotzdem qualifiziert sich die Shell als Testumgebung für verschiedene Experimente mit Algorithmen aus vielen Bereichen der künstlichen Intelligenz.

Über verschiedene Konfigurationsdateien für die A–Box, T–Box und die Operatorendatenbank sowie die Tools und damit verbundene Konfigurationsdateien für die Klassifikatoren sind neue Domänen schnell und einfach zu erfassen. Leider sind die Konfigurationsdateien jeweils in einem proprietären Format verfaßt. Es wäre sinnvoller gewesen sie z.B. in einer XML-Instanz aufzuschreiben, damit wäre eine Übernahme der Daten aus anderen und in andere Anwendungen evtl. vereinfacht worden. Schade ist nur, daß es keine entsprechende Beschreibungssprache für die Gestaltung der grafischen Oberfläche gibt. Das wäre eine weitere Vereinfachung der Shellbenutzung gewesen.

Durch die angesprochenen Konfigurationsdateien sowie die globale Konfigurationsdatei für einen Agenten können zahlreiche Einstellungen vorgenommen werden, um den Lauf eines Agenten zu optimieren. Einerseits hinsichtlich seiner Geschwindigkeit und andererseits hinsichtlich der Qualität der Ergebnisse. So kann z.B. die Bewertung der Operatoren auch ohne einen Operatorlerner vom Shellbenutzer selbst einfach mittels einer Textdatei (s. II 8) vorgenommen werden, um die Pläne des `PlannerTypeA` zu verbessern.

6.3 Ausblick

Trotz der Vielzahl der standardmäßig implementierten Funktionen der Shell sind noch zahlreiche Erweiterungsmöglichkeiten vorhanden.

Zuerst sind die Klassen und Funktionen zu nennen, die die PG zunächst in ihrem Entwurf vorgesehen hatte, aber aus Zeitgründen nicht mehr implementieren konnte. Das sind der Operatorenlerner (s. II 15) und ein SVM–Klassifikator.

Die nächsten Erweiterungen stellen Verbesserungen der Basisimplementierungen der BotIshellys dar. Es sind z.B. Planer, die komplexere Pläne erstellen als nur lineare Listen und Planausführer die auch mit mengenwertigen mehrstelligen Prädikaten umgehen können denkbar. Benutzerprofile, statistisch belegbarere Rankingmodelle und unschärfere logische Abfragen sind weitere Ideen, durch die die Shell sinnvoll ergänzt werden könnte.

Die hier angeregten Verbesserung bzw. Ergänzungen sind zum Teil bewusst von der PG von vorneherein als Ergänzung zu ihrer Arbeit betrachtet worden und zum Teil durch die Erfahrungen während der Implementierung bzw. während der Benutzung BotIshellys zur Implememtierung eines Agenten als wünschenswert empfunden worden.

Im folgenden werden einige Ergänzungsmöglichkeiten näher beschrieben. Wenn die PG schon weitergehende Überlegungen zu der vorgeschlagenen Erweiterung angestellt hat, dann werden sie kurz erläutert, um einen Aufsatzpunkt für die Erweiterung zu bieten.

6.3.1 Operatorenlerner

Ein Operatorenlerner, der eine Bewertung der Agentenoperatoren vornimmt und dadurch den Planern die Auswahl „guter“ Operatoren erlaubt oder Operatoren zusammensetzt und diese in die Operatordatenbank einfügt, könnte die Qualität der generierten Pläne verbessern. Die PG hat die Implementierung dieses Lerners zurückgestellt, da er nicht für die Basisfunktionalität der Shell benötigt wird. Die Pläne des `PlannerTypeA` verlieren dadurch jedoch an Güte, da diesem Planer damit die Orientierung bzgl. der Operatorenauswahl fehlt.

Die Überlegungen, die die PG angestellt hat, haben ergeben, daß die Implementierung eines Lerners, der neue Operatoren aus den schon vorhandenen zusammensetzt nicht trivial ist, da zu klären ist wie aus den Vor- und Nachbedingungen der Einzeloperatoren die Vor- und Nachbedingung des Gesamtoperators zu erstellen sind. Die PG ist mit ihren Überlegungen bis zu folgendem Punkt gelangt:

Die Vorbedingung der neuen Operatoren sollen aus den Vor- und Nachbedingungen der Operatoren, die in dem Teilplan verwendet werden, synthetisiert werden.

Beispiel: $\{V_1, V_2, V_3\}$ und $\{N_1, N_2, N_3\}$ seien die Vor– bzw. Nachbedingungen der drei Operatoren OP_1, OP_2, OP_3. Die Vorbedingung des neuen Operators OP_{neu} sei dann

$$V_{neu} := ((V_1 \cup V_2)\backslash N_1) \cup V_3)\backslash N_2$$

Oder in Worten: die Vorbedingungen aller drei Operatoren werden vereinigt und die Vorbedingungen, die durch die Nachbedingungen der vorangegangenen Operationen erfüllt werden, werden abgezogen.

Die Nachbedingungen Die Nachbedingung des neuen Operators bildet sich aus der Vereinigung der Operatoren des Teilplans.

Beispiel: Unter den Voraussetzungen von oben

$$N_{neu} := N_1 \cup N_2 \cup N_3$$

Leider funktioniert diese recht einfache Behandlung der Vor- und Nachbedingung nur bei Operatoren, deren Delete–Listen leer sind und die genau eine Nachbedingung haben. Für den allgemeinen Fall mit mehreren Nachbedingungen und Delete–Listen müssen komplexere Lösungen gefunden werden. Das Problem der gelöschten Prädikäte muß beachtet werden und es muß geklärt werden was passiert, wenn innerhalb des zusammengesetzten Operators ein Operator nicht die gewünschte Nachbedingung erfüllt. Dafür muß evtl. ein komplexerer `CompoundOperator` geschrieben werden. Die aktuelle Implementierung unterstützt nur lineare Operatorenschachtelungen.

Die PG konnte keine zufriedenstellende Lösung für den allgemeinen Fall finden.

6.3.2 Klassifikatoren

Zusätzlich zu den schon implementierten Rocchio-, Bayes- und k-NN-Klassifikatoren können beliebige andere Textklassifikatoren in die Shell eingebunden werden. Speziell erwähnt sei hier nochmal die SVM (s. [Vapnik, 1998], [Joachims, 1997]) von der sich einige PG-Teilnehmer verbesserte Klassifikationsergebnisse versprechen. Auch Bildklassifikatoren könnten in BotIShelly ergänzt werden, aber für diese sind die Vorarbeiten nicht annähernd so umfangreich wie für Textklassifikatoren.

Bei der praktischen Anwendungen fiel auf, daß es sinnvoll zu sein scheint, nicht nur über Textklassifikation durch Wortvektorähnlichkeit zu klassifizieren, sondern daß es z. B. bei der Suche nach Produktseiten besser wäre, nach ähnlichen Seitenstrukturen zu klassifizieren.

Bild- und Tondokumente

Die Klassifikation von Bild- und Tondokumenten ist zwar nicht so weit fortgeschritten wie die Klassifikation von Texten, jedoch schien es der PG immer wichtiger zu werden, Textextraktion aus Bildern in Erwägung zu ziehen. Immer mehr Firmen schmücken ihre Seiten mit grafischen Firmen–Logos oder Informationen, in denen der gesuchte Firmenname steht, und davon absehen den Firmennamen nochmal textuell anzugeben. Dasselbe gilt für viele Produkte. Die Idee der Tondokumenterkennung scheint zum gegenwärtigen Zeitpunkt zwar noch nicht notwendig, ist aber aufgrund des Wunsches nach universeller Einsetzbarkeit BotIshellys genauso wie die Bildklassifikation prototypisch vorgesehen und kann daher, Kenntnis entsprechender Algorithmen vorausgesetzt, einfach ergänzt werden.

Symbolische NLP-Methoden

Zur Absicherung bzw. Unterstützung statistikbasierter Textklassifikatoren können Techniken der symbolischen Verarbeitung natürlicher Sprache eingesetzt werden. Methoden der Satz- und Textanalyse sollten die Verläßlichkeit der Ergebnisse der Textklassifikation weiter steigern.

6.3.3 Planer

Die implementierten Planertypen A und C verwirklichen zwei Planer, die exemplarisch zeigen sollen, wozu die Shell in der Lage ist. Der `PlannerTypeA` ist dabei als primitiver Testplaner gedacht und der `PlannerTypeC` ein Versuch, die Unwägbarkeiten des Internets in einem klassischen Planungsansatz (s. I A.7) zu berücksichtigen. Gerade hier sind noch viele Erweiterungs- und Verbesserungsmöglichkeiten zu finden. Es ist z.B. zu überlegen, wie die dynamische Welt von einem Planer noch besser zu bewältigen ist oder wie ein Planer kleinere Operatoreinheiten in vernünftiger Laufzeit bewältigen kann. Zusätzlich kann BotIshelly als Testplattform für andere (literaturbekannte) Planer verwendet werden, um experimentell zu erforschen, wie sich diese Planer in der sich ständig ändernden Welt des Internets zurechtfinden. Aufbauend auf den von der PG entwickelten Planer Typ C, scheint es ein vielversprechendes Projekt zu sein, einen Planer zu entwickeln, der verschiedene Wege zum Ziel in einem Planbaum an den Planausführer übergibt. Ein weiterer Ansatzpunkt zur Weiterentwicklung ist zu berücksichtigen, welche Operatoren mit welchen Vorbedingungen schon vom Planer benutzt wurden. So kann das doppelte Erzeugen von dynamischen Objekten, die eigentlich aus derselben Operatorkonstellation entstanden sind vermieden werden. Generell ist das Planen in einer Internetwelt ein Gegenstand der aktuellen Forschung (s. das DFG–Projekt [CAPlan, 1998]), für den BotIshelly eine interessante Testumgebung bietet.

6.3.4 Planausführung

Die Planausführung kann in ihrer jetzigen Implementierung nicht mit mengenwertigen mehrstelligen Prädikaten umgehen. Es ist zu überlegen, wie diese Funktionalität, die im Zusammenhang mit dem Internet und den von der PG eingeführten dynamischen Objekten wichtig erscheint, noch eingebracht werden kann. Dabei sind die Probleme, die sich durch die Verzeigerung der Operatorenvor- bzw. -nachbedingungen ergeben zu beachten. Eine Änderung der Menge pflanzt sich dadurch durch alle Operatoren, die dasselbe Prädikatsargument benutzen durch. Das kann zu unsinnigen Operatorbelegungen führen und zu einem merkwürdigem Backtrackingverhalten des Planausführers, da die Mengenänderung sich auch auf schon ausgeführte Operatoren auswirkt (s. I 6.2.2).

Die Heuristik, die Instanzen für nicht belegte Prädikate auswählt, sollte verbessert werden.

6.3.5 Visualisierung von Plänen

Im Laufe der Arbeit mit der Shell und zugegebenermaßen der Fehlersuche während der Implementierung, wäre es wünschenswet gewesen, sich die Planungslevel des `PlannerTypeC` grafisch anschauen zu können, um Verbesserungsmöglichkeiten zu finden. Dies würde auch die Fälle zu verstehen helfen, in denen kein Plan generiert werden konnte.

6.3.6 Rankingmodelle

Die Ergebnisse eines Agenten werden bei der Ausgabe in irgendeiner Form sortiert ausgegeben. Im allgemeinen erwartet der Benutzer, daß die Ergebnisse nach Relevanz sortiert sind. Der von der PG implementierte Testagent verwendet ein einfaches Rankingverfahren, das auf den Ergebnissen der Klassifikation durch die Klassifikatoren und die Operatoren basiert. Hier sind noch andere Ansätze und Rankingmodelle vorstellbar. Diese können mit BotIShelly ausprobiert und evaluiert werden.

6.3.7 Benutzerprofile

Im Zusammenhang mit dem Ranking ist während der Modellierung verschiedentlich die Frage gestellt worden, ob es sinnvoll wäre, Benutzerprofile zu erstellen und davon ausgehend die Ergebnisse zu bewerten. Bedingt durch zeitliche Beschränkungen und den datenschutzrechtlichen Bedenken einiger PG–Mitglieder wurden die Benutzerprofile nicht direkt implementiert, aber sie sind durch einige Änderungen in der Systemkontrolle und der Rankingmethode in einer Subklasse von `PlanInformation` leicht ergänzbar.

6.3.8 Unscharfe Anfragen

Die Eingabe eines Benutzers oder einer Benutzerin in den Agenten muß in einen logischen Ausdruck umgewandelt werden, den der Planer versteht. Dieser versteht aus Komplexitätsgründen (s. I A.7) nur Konjunktionen ohne Verneinung. Andererseits könnte diese Anfrage möglichst allgemein gehalten werden, so daß der Agent viele Seiten findet, die dann anschliessend durch eine entsprechende verify-Methode in `PlanInformation` gefiltert werden. Hier können komplexere Logiken zum Einsatz kommen, um einer möglichst großen Nutzerschaft zu genügen.

6.3.9 GUI

Im Moment zielt die Implementierung der Benutzungsoberfläche auf den Einsatz von Servlets und Browsern. Es ist jedoch ebenso möglich, die I/O durch eine eigenständige Java–Swing–Oberfläche zu ersetzen. Außerdem kann über eine Beschreibungssprache für die grafische Oberfläche nachgedacht werden, die aus einer Konfigurationsdatei eine grafische Standardoberfläche mit den vom Shellbenutzer gewünschten Knöpfen und Feldern erzeugt.

6.3.10 Format der Konfigurationsdateien

Die proprietären Sprachen der Textdateien, die zur initialen Füllung von T–Box, A–Box und Operatordatenbank dienen, sollten in eine XML-Instanz umgewandelt werden, um vor allem im Falle der T– und A–Box die Datensätze auch in oder aus anderen Anwendungen nutzen und darstellen zu können.

6.4 Fazit

Wir haben gezeigt, daß es möglich ist, aus der Shell Suchagenten zu erstellen. Die Shell stellt die Grundbausteine zur Verfügung und entlastet den Shellbenutzer damit von der Implementierung z. B. von Schnittstellen zum Web, Textklassifikatoren, Planer etc. Der Anpassungsaufwand beschränkt sich daher auf den domänenabhängigen Teil des konkreten Agenten. Eine gute und angemessene Modellierung des Sachbereichs ist ausschlagebend für den Erfolg des Agenten. Der Aufwand hierfür ist nicht zu unterschätzen.

In Anbetracht der Tatsache, daß der Firmen–Agent nur einen Prototyp darstellt, sind die Ergebnisse relativ gut. An vielen domänenspezifischen Stellen ist dabei Raum für Verbesserungen vorhanden, vor allem im Bereich der Operatoren.

Abschliessend kann man feststellen, daß der Shellansatz erfolgreich war. Man kann sich zum Beispiel vorstellen, daß mehrere Agenten für verschiedene Anwendungen erstellt werden und darüber arbeitet ein „Superagent“, der Anfragen auf die anderen Agenten verteilt. So könnte man einen komplexen, universalen Suchagenten realisieren.

Anhang A

Ausarbeitung der Seminarvorträge

A.1 Was ist ein Agent?

von Jens Jägersküpper

A.1.1 Einleitung

Der Begriff des Agenten ist für den Bereich Künstliche Intelligenz (KI) von zentraler Bedeutung, da manche der Auffassung sind, daß die KI das Teilgebiet der Informatik sei, das sich mit der Konstruktion von Agenten, welche Aspekte von intelligentem Verhalten an den Tag legen, beschäftigt. Um so erstaunlicher, daß bisher der Begriff des Agenten nicht einmal innerhalb der KI einheitlich aufgefaßt wird, geschweige denn formal definiert ist. Dies hängt sicherlich auch damit zusammen, daß ebenso Uneinigkeit darüber besteht, was denn nun intelligentes Verhalten — bzw. Intelligenz an sich — überhaupt ist.

Im folgenden soll trotzdem versucht werden, Agenten ein wenig genauer auf die Spur zu kommen. Dabei orientiere ich mich an den Artikeln *„Intelligent Agents: Theory and Practice"* von Michael Wooldridge und Nicholas Jennings [Wooldridge and Jennings, 1995] und *„Formalizing Properties of Agents"* von Richard Goodwine [Goodwin, 1993].

A.1.2 „Der kleinste Nenner"

Zunächst ein schwacher Begriff für einen Agenten, dem jedoch sehr viele — nicht nur im Bereich der KI — zustimmen. Es handelt sich eigentlich nur um eine Auflistung von Eigenschaften, die ein Stück Hardware oder ein Programm, das in einer irgendwie gearteten Umgebung abläuft, aufweisen sollte, damit man von einem Agenten sprechen kann.

Autonomie Agenten sollten (nach ihrer Aktivierung) ohne direkte Führung durch ihren Auftraggeber oder andere ihre Aufgaben bewältigen. Sie sollen ihr Handeln also selbst kontrollieren.

Sozialverhalten Agenten sollten mit anderen Agenten oder auch Menschen auf irgendeine Art, z. B. mit einer besonderen Sprache, kommunizieren können.

Reaktionsfähigkeit Agenten müssen ihre Umwelt (z. B. die physikalische Welt, eine graphische Benutzerschnittstelle, das Internet o. ä.) wahrnehmen können und auf etweige Änderungen **in einer angemessenen Zeit** reagieren können, falls dies notwendig ist.

„selbstbestimmtes" Agieren Andererseits sollte das Handeln von Agenten nicht nur durch „äußere Reize" bestimmt werden. Vielmehr sollten sie selbst die Initiative ergreifen und zielgerichtetes Verhalten entwickeln, um ihren Aufgaben **in angemessener Zeit** nachzugehen.

Diese umgangssprachliche Beschreibung von vermeintlich charakteristischen Eigenschaften läßt naturgemäß viel Platz für Interpretation — wahrscheinlich ein Hauptgrund, warum sich sehr viele mit diesen Charakteristika anfreunden können. Demnach ist durchaus in einem Auto mit Anti-Blockier-System, Anti-Schlupf-Regelung und elektronischem Stabilitätsprogramm ein Dreiergespann von Agenten aktiv.

A.1.3 Ein stärkerer Begriff eines Agenten

Manch anderem schwebt im Zusammenhang mit (künstlichen) Agenten etwas anderes vor, z. B. ein persönlicher, digitaler Assistent (PDA), der u. a. folgendes leisten könnte:

Durch das *Beobachten* seines „Meisters" z. B. bei der Recherche im WWW *weiß* bzw. *lernt* der Agent, wofür sich dieser interessiert. Daraufhin sucht er *selbständig* nach Artikeln und offeriert diese seinem Nutzer. Darüberhinaus durchforstet und sortiert bzw. trennt der PDA noch die E–mails nach privaten und beruflichen und/oder wichtigen und unwichtigen. So entscheidet er, welche eingehenden Nachrichten sofort „durchgestellt" werden und welche für eine „ruhige Minute" aufgehoben werden. Natürlich ist der PDA auch an der Terminplanung beteiligt und macht bei auswärtigen Terminen selbständig Angaben über die besten Flug-, Zug- oder Sonst–wie–Verbindungen und deren Kosten, nachdem er die entspechenden Transportunternehmen kontaktiert hat. Auch eine eventuelle Buchung wird vom Nutzer nur noch „angestoßen" und dann durch den PDA getätigt. Anschließend kümmert sich der Agent in gleicher Weise auch noch um eine passende Bleibe.

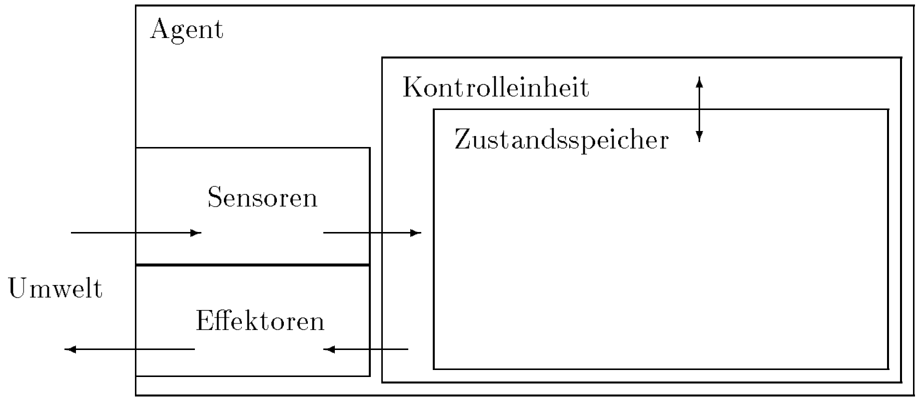
Wie man schon an der visionären Beschreibung des obigen PDA bemerkt, werden dem Agenten Fähigkeiten und Eigenschaften zugerechnet, die man normalerweise in bezug auf Menschen benutzt. In dem durch das Beispiel verdeutlichten, „stärkeren" Zusammenhang ist ein Agent ein Computer–System, das zusätzlich zu den oben erwähnten Eigenschaften in seiner Konzeption oder Implementierung Konzepte aufweißt, die normalerweise Anwendung auf Menschen bzw. deren Eigenschaften und Fähigkeiten finden. Üblich sind „mentale" Begriffe wie z. B. „wissen", „annehmen", „Überlegungen anstellen", „eine Absicht haben" u. ä.. Es gibt sogar Überlegungen zu „emotionalen" Agenten.[1]

A.1.4 Grundstruktur und -begriffe eines Agenten

Ob Agent in schwachem oder in starkem Sinne, um seinen Aufgaben nachzukommen, muß ein Agent i. a. einen **Mechanismus zur Interaktion** (inkl. Kommunikation) mit seiner Umwelt besitzen. Aus diesem Grund verfügt ein Agent über **Sensoren**, mit denen er seine Umwelt wahrnimmt, und über **Effektoren**, die ihm dazu dienen, in seiner Umgebung „Spuren zu hinterlassen" oder auch im engeren Sinne zu kommunizieren. In obigem Auto–Beispiel sind u. a. Sensoren für die Drehzahl der Räder, die Lengs- und Querbeschleunigung vorhanden. Genauso stellt aber auch das Bremspedal einen Sensor dar. Effektoren sind u. a. Bremskrafterzeuger, -regler und -kolben, aber auch Kontrollanzeigen, die der (Einweg-) Kommunikation mit dem Fahrer dienen.

Den „Rumpf" und/oder das „Gehirn" des Agenten bildet die **Kontrolleinheit**, die die Signale der Sensoren verarbeitet und die Effektoren steuert. Sie besitzt einen **Zustandsspeicher**, in dem Informationen gehalten werden, die den internen Zustand des Agenten widerspiegeln, aber genauso gut auf die Umwelt bezogen sein können. In unserem Auto–Beispiel sind dies ein oder ein paar kleine Kästchen, in die viele Kabel und Leitungen führen.

[1]Wie auch immer diese „aussehen" sollen...



Grundlegende Begriffe im Zusammenhang mit Agenten sind:

effektorisch kompetent Ein Agent ist kompetent bzgl. des Agierens in seiner Umwelt (engl.: *capable*), wenn er über Effektoren verfügt, die ihm die Bewältigung seiner Aufgaben ermöglichen.

sensorisch kompetent Sind die Sensoren eines Agenten so beschaffen, daß ihm mit diesen die Bearbeitung seiner Aufgaben möglich ist, so ist er bzgl. der Wahrnehmung seiner Umwelt kompetent (engl.: *perceptive*).

kompetent/angepaßt ist ein Agent, wenn sein Interaktionsmechanismus geeignet ist, die Bearbeitung seiner Aufgaben zu bewerkstelligen; er also über effektorische und sensorische Kompetenz verfügt.

erfolgreich Damit ein Agent Erfolg[2] im Sinne der Lösung bzw. Abarbeitung seiner Aufgaben haben kann, ist Kompetenz eine Notwendigkeit, da es sich sonst nur um „Zufallstreffer" handeln kann.

Es ist zu beachten, daß das Zutreffen der obigen Begriffe von der Umwelt abhängt. Es handelt sich also um relative Begriffe, so daß bei nicht eindeutig klarer Relation zur Umwelt Erläuterungen nötig sind. Beispiel: Ein Textklassifizierungssystem, das (mehr oder minder selbständig und erfolgreich) ASCII–Texte bearbeitet, versagt bei als Grafik abgespeicherten Texten, da er dieses Format nicht kennt oder erkennt. Es ist also bei „widrigen Umweltbedingungen" nicht erfolgreich, da seine sensorische Kompetenz nicht mehr gegeben bzw. ausreichend ist.[3]

Nun aber zu einem klassischen Agenten–Schema, dessen Bezeichnung sich vom lateinischen Wort „deliberatio" herleitet, welches mit „Beratschlagung" oder auch „Überlegung" zu übersetzen ist.

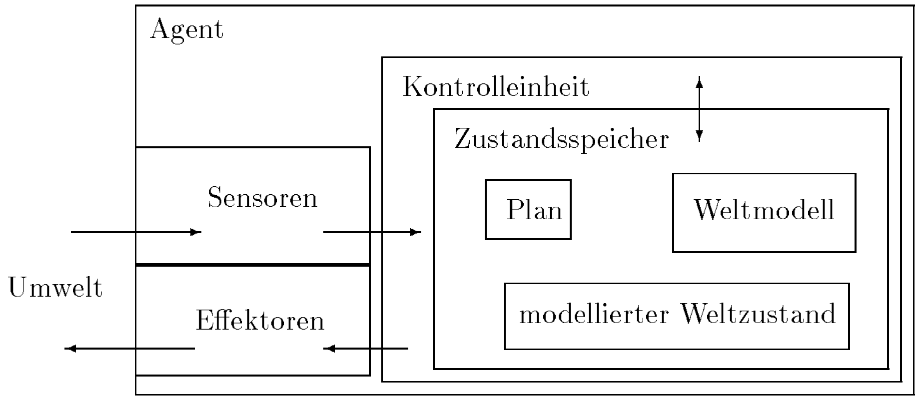
A.1.5 Klassisch: Der deliberative Agent

Als deliberativ bezeichnet man einen Agenten, der eine explizite Repräsentation der Welt durch Symbole benutzt, und der seine Entscheidungen (z. B. über seine Aktionen) durch logisches, zumindest aber pseudo–logisches Schließen tätigt, welches auf dem Vergleich und der Manipulation von Symbolen basiert. Diese Art Agenten nutzen also durch Symbole repräsentiertes Wissen über ihre Umwelt, das ihnen „eingepflanzt" wurde und sie mit Hilfe ihrer Sensoren erlangen bzw. erweitern und aktualisieren.

[2] Die Frage, ob ein Agent denn nun Erfolg gehabt hat oder nicht, ist häufig durchaus nicht trivial.

[3] Deises Beispiel ist vielleicht nicht ganz so schön, da der Mißerfolg auch an einer „unfähigen" Kontrolleinheit liegen kann. Dadurch wird aber deutlich, daß die Suche nach Gründen für Erfolglosigkeit ebenfalls nicht–trivial sein kann. Ist die Kompetenz des Agenten in einer bestimmten Umwelt gezeigt (besser noch bewiesen), kann es nur noch an „fehlender oder falscher Kontrolle" liegen.

Durch das Modellieren seiner Umwelt durch Symbole hat der Agent ein ganz bestimmtes Bild der Welt, welches sich in dem von ihm angenommenen, modellierten Zustand der Welt widerspiegelt. Auf dessen Grundlage entscheidet er über seine Aktionen, was man im weiteren Sinne „planen" nennen kann.



Ein Grund für diese Konstruktion ist die Hypothese über Systeme, die auf physischen Symbolen basieren (engl.: *physical-symbol system hypothesis*). Ein solches System besteht aus einer physisch realisierbaren Menge von Symbolen, die zu Strukturen kombiniert werden können, und aus einer Menge von durch Symbole kodierten Regeln bzw. Instruktionen, die in einem laufenden Prozess auf obige Symbole angewendet werden können. Die von Newell und Simon 1976 formulierte Hypothese besagt dann, daß ein solches System das Potenial für generelle Intelligenz besitzen kann.

Zu den üblichen Begriffen, die im Zusammenhang mit Agenten benutzt werden, kommen bei deliberativen Agenten zu denen aus dem vorigen Kapitel noch weitere hinzu:

vorhersagend (engl.: *predictive*) Ein Agent besitzt diese Eigenschaft, wenn seine Modellierung der Welt derart beschaffen ist, daß er beim Abwägen darüber, ob eine bestimmte Aktion geeignet ist, ihn weiter an sein Ziel zu bringen, die richtige Erkenntnis erlangt (d. h. die, die mit der echten Welt konsistent ist).

interpretativ (engl.: *interpretive*) Dies trifft zu, wenn ein Agent die Daten, die er von seinen Sensoren übermittelt bekommt, konsistent zur (echten) Welt interpretiert.

rational Ein Agent ist rational, wenn er sich für Aktionen entscheidet, die er für geeignet hält, d. h. in seinem Weltmodell zum Erfolg führen.

vernünftig (engl.: *sound*) Ist ein Agent vorhersagend, interpretativ und rational, so spricht man von einem vernünftigen Agenten.

Dieser Ansatz klingt sehr verlockend und einfach: Zum Modellieren der Welt benutzt man eine der „gängigen" Logiken und dann „spendiert" man dem Agenten noch einen irgendwie gearteten automatischen Beweiser, der für das Entscheiden mit Hilfe von logischem Schlußfolgern zuständig ist. Dabei einen rationalen Agenten zu erhalten ist nicht schwer. Damit aber auch ein vernünftiger Agent dabei herumkommt, sind mindestens drei Hauptprobleme zu beachten:

1. Das Transduktionsproblem besteht darin, die Welt akkurat und adequat durch Symbole zu beschreiben, so daß das Weltmodell es dem Agenten ermöglicht, vorhersagend und interpretativ zu sein. (Außerdem sollte das Erzeugen dieses Modells nur absehbar lange dauern und auch sonst nur realistische Ressourcen benötigen, damit der Agent auch irgendwann zum Einsatz kommen kann.)

2. Auch mit dem logischen Schließen gibt es ein Ressourcen–Problem. Ein Agent handelt immer in einem zeitlichen Rahmen, und wie schon im ersten Kapitel erwähnt, sollte er diesen möglichst nicht verlassen, d. h. u. a. in angemessener Zeit auf ein Ereignis reagieren können. Da schon die Prädikaten–Logik erster Stufe unentscheidbar ist, kann der automatische Beweiser u. U. zu gewaltigen Problemen in dieser Hinsicht führen.

3. Das Aktualisieren des Weltmodells erfordert Algorithmen zur Symbol–Manipulation, die auch in vernünftiger Zeit arbeiten müssen. Auch für diese haben sich die Zeitanforderungen als kritisch herausgestellt.

Derartige Probleme der symbol–basierten KI stellten sich in der Praxis als derart schwierig und hartnäckig heraus, daß einige Leute, u. a. Rodney Brooks vom MIT, nach Alternativen gesucht haben.

A.1.6 Alternativ: Reaktive Agenten–Architektur

Ein Agent basiert auf einer reaktiven Architektur, wenn er nicht über ein zentrales, durch Symbole realisiertes Weltmodell verfügt und seine Schlußfolgerungen auch nicht anhand komplexer Symbolmanipulation tätigt.[4] Brooks, der an autonomen mobilen Robotern arbeitete, hat die folgenden Thesen aufgestellt:

1. Intelligentes Verhalten ist auch *ohne* explizite Repräsentation bzw. Modellierung der Welt möglich.

2. Intelligentes Verhalten kann auch *ohne* explizites abstraktes Schlußfolgern, wie es die symbol–basierte KI vorschlägt, erzeugt werden.

3. Intelligenz ist eine sich bei bestimmten komplexen Systemen ergebende Eigenschaft.

Brooks stellte zwei Kernideen bzw. -einsichten heraus, die ihn zu diesen Thesen leiteten:

1. „Echte“ Intelligenz ist nicht in entkörperlichten Systemen wie automatischen Beweisern oder Experten–Systemen enthalten, sondern entwächst der Welt.

2. „Intelligentes Verhalten“ ist ein Ergebnis bzw. eine Folge der Interaktion des Agenten mit seiner Umwelt. Ebenso liegt die Intelligenz im Auge des Betrachters und ist nicht eine isolierte, absolute, wesensmäßige Eigenschaft.

Die „subsumption architecture“ von Brooks

Für seine Roboter benutzte Brooks eine hierarchische Anordnung von Verhaltenskonzepten, die sich im Wettstreit um die Kontrolle über den Roboter befinden. Die Verhaltenskonzepte auf unteren Ebenen sind dabei für „primitives“ Verhalten, wie das Ausweichen eines Hindernises, zuständig, und haben Vorrang vor komplexeren Verhaltenskonzepten darüberliegender Ebenen. Auch wenn Brooks Systeme extrem simpel gewesen sind, so sollen sie teilweise Verhalten gezeigt haben, die für „konventionelle“, symbol–basierte Systeme erstaunlich gut gewesen wären. Dies lag sicherlich auch an besseren Antwortzeiten seiner Systeme gegenüber den „langwierig abwägenden“ Agenten klassischer Art.

[4]Der Begriff „reaktiv“ bezeichnet auch eine ganz bestimmte Eigenschaft eines Agenten im allgemeinen Zusammenhang, d. h., wenn die ihm zugrunde liegende Architektur nicht betrachtet wird.

Die „situated automata" von Rosenschein und Kaelbling

Ein weiterer, raffinierter Ansatz kommt von Rosenschein und Kaelbling. In ihrem „situated automata" Paradigma wird ein Agent in Form von Deklarationen beschrieben. Diese Spezifikation wird dann zu einer digitalen Maschine kompiliert, die den deklarierten Spezifikationen genügt. Für die dabei entstehenden digitalen Maschinen können Antwortzeiten garantiert bzw. bewiesen werden. Sie führen keine Symbolmanipulationen aus und es sind in ihnen auch keine symbolischen Ausdrücke repräsentiert. Die Spezifikation ist zweigeteilt und es bezieht sich ein Teil auf die Verarbeitung der sensorischen Reize und der andere auf das Agieren des Agenten. Zur Synthetisierung des Agenten werden dann zwei Programme benutzt, die auf den beiden Teilen der Spezifikation arbeiten:

Das Programm RULER ist für die Reizverarbeitung zuständig und hat drei Eingabekomponenten:

1. Die Spezifikation der Semantik der Sensorendaten (z. B. „wenn dieses Bit gesetzt ist, regnet es")

2. Eine Menge von statischen Fakten (z. B. „wenn es regnet, ist der Boden naß")

3. Ein Menge von Regeln, die die Änderung der Welt berücksichtigen (z. B. „wenn der Boden naß ist, bleibt er feucht, bis die Sonne rauskommt")[5]

Der Programmierer spezifiziert dann die Semantik des Ergebnises der Reizverarbeitung (z. B. „wenn dieses Bit gesetzt ist, ist der Boden naß") und der Compiler synthetisiert daraus einen Schaltkreis, der diese Semantik widerspiegelt. Das ganze „deklarative Wissen" wird also auf einen gewöhnlichen Schaltkreis reduziert.

Das Programm GAPPS hat als Eingabe eine Menge von Regeln, die das erreichen des Ziels beschreiben bzw. Informationen, die beschreiben, wie das Ziel zu erreichen ist (engl.: *goal reduction rules*). Daraus erzeugt GAPPS ein Programm, welches das Agieren des Agenten kodiert und in einen digitalen Schaltkreis übersetzt werden kann. In dem Schaltkeis sind wie gesagt keine Symbol–Repräsentationen enthalten und jegliche Symbolmanipulation findet während des Kompilierens statt.

Agenten Netzwerke von Maes

Bei dieser Architektur ist ein Agent als eine Menge von „Kompetenz–Modulen" definiert, die mit den Verhaltenskonzepten der Brooks'schen „subsumption architecture" vergleichbar sind. Jedes Modul wird durch Vor- und Nachbedingungen beschrieben und besitzt ein „Aktivierungslevel", der die Relevanz des Modules in bestimmten Situationen festlegt. Je größer dieser Wert ist, desto wahrscheinlicher ist es, daß das Modul das Verhalten des Agenten in der entsprechenden Situation beeinflußt.

Sind die Module einmal spezifiziert, so werden sie zu einem Netzwerk von Modulen (engl.: *spreading activation network*) — ähnlich einem neuronalen Netz — kompiliert, dessen Verbidungen durch die Vor- und Nachbedingungen bestimmt sind.

Beim Ausführen des Agenten werden dann je nach Situation bestimmte Module aktiviert und werden daraufhin ausgeführt. Das Ergebnis des Ausführens eines Moduls kann eine bestimmte „externe Aktion" mit Hilfe der Effektoren sein, aber auch netzinterne Veränderungen, wie das Verändern von Aktivierungsleveln einiger Module o. ä..

[5] Wo lebt denn derjenige, der sich das Beispiel ausgedacht hat?

A.1.7 Integrativ: Hybride Agenten–Architektur

Bei hybriden Systemen wird versucht, die Vorteile von deliberativen Agenten mit denen von Agenten, die eine reaktive Architektur besitzen, zu kombinieren. Pate stand (natürlich) wieder der Mensch, der eine gewisse Abstufung im Einsatz von Intelligenz bei bestimmten Abläufen erkennen läßt:

Der Mensch besitzt **Reflexe**, bei denen das Gehirn nicht beteiligt ist, also keine „echte“ Intelligenz gebraucht wird. Auf der nächsten Stufe sind **stereotype Verhaltensmuster** angesiedelt, die der Mensch zwar erst einmal erlernen muß, wofür gewisse Intelligenz nötig ist, die dann aber ohne große Anteilnahme des Gehirns ablaufen. Die oberste Stufe bilden **„bewußte“ Prozesse**, auf die sich der Mensch konzentrieren muß und bei denen er „seine ganze Intelligenz demonstrieren“ kann.

Reflexe und stereotype Verhaltensmuster sind Eigenschaften, die man unschwer bei der reaktiven Architektur findet. Für bewußte Prozesse scheinen die dem deliberativen Agenten zugrunde liegenden, symbol–basierten Mechanismen besser geeignet zu sein. Dies führt zu einer „Schicht–Architektur“, bei der die (Kontroll–) Subsysteme des Agenten hierarchisch geordnet sind bezüglich Abstraktion (von der echten Welt in die modellierte). Entlang der Hierarchie nimmt der Abstraktionsgrad zu — und damit hoffentlich auch die Intelligenz — dafür aber die Reaktionsgeschwindigkeit ab. Die unteren Schichten stellen dabei reaktive Machnismen zur Verfügung, die bei Umweltbedingungen greifen, die eine schnelle Reaktion erfordern, z. B. das Ausweichen eines Hindernises. Die oberste Schicht ist mit dem Planen und Überwachen von langfristigen Strategien beschäftigt (die auf das Erreichen des Ziels ausgerichtet sind).

Als „Knackpunkt“ stellt sich bei dieser Schicht–Architektur der Kontrollfluß zwischen den Schichten heraus, der die Interaktion zwischen den einzelnen Subsystemen realisiert und dafür sorgen soll, daß der Agent als Ganzes ein vernünftiges Verhalten zeigt.

Beispiele für diese Architektur sind die von Ferguson vorgeschlagenen *„Touring–Machines“* und *„InterRap“* von Müller und Pischel.

A.2 Softbots

von Michael Banken

A.2.1 Einleitung

Die Internet Softbots (Software Roboter) wurden 1993 an der Universität von Washington von Etzioni sowie Lesh und Segal entwickelt. Der Softbot nutzt eine Unix Shell und das Internet mit einer Menge von Internet Ressourcen.

- Effektoren

  FTP

  Telnet

  Mail

  Datei manipulierende Funktionen

- Sensoren

  Archie

  Gopher

  Netfind ...

Die Bedeutung des Softbots läßt sich an drei Punkten festmachen:

- Es wird ein integriertes und ausdrucksvolles Interface zum Internet unterstützt.
- Der Softbot besitzt die Fähigkeit von einer Umgebung in die andere zu wechseln, soweit dies der Auftragserfüllung dienlich ist.
- Der Softbot wählt dynamisch, welche Möglichkeiten er nutzten möchte und in welcher Sequenz.

A.2.2 Softbot-unterstütztes Interface

Ein Softbot sollte sich wie ein persönlicher Assistent verhalten. Natürlich kann ein Mensch die ihm aufgetragenen Aufgaben auf einem viel höheren Entwicklungsgrad durchführen. Der Softbot muß diesen Nachteil ausgleichen, indem er alle ihm zur Verfügung stehenden Hilfsmittel intelligent einsetzen kann. Der Softbot behandelt Ziele nach einer 'First Order Logic'. Dies bedeutet, es können Konjunktion, Disjunktion, Negation und universelle Quantifizierung kombiniert werden. Da solche Notationen für ungeübte Benutzer sehr kompliziert erscheinen, sollten hier Menüsysteme die Eingabe erleichtern und automatisch eine Übersetzung in die Logik beinhalten. So gesehen kann jede Art des Dialogs mit dem Softbot genutzt werden, solange ein Modul für die Übersetzung existiert.

A.2.3 Interface Design Prinzipien

Bei dem Design der Schnittstelle, wurde sehr viel Wert auf die Flexibilität gelegt. Folgende Ideen sollten berücksichtigt werden:

- ZIELORIENTIERT

  Eine Anfrage indiziert, was der Benutzer von dem Softbot möchte. Der Softbot muß nun entscheiden, wie und wann er dieser Anfrage nachgeht.

- TOLERANT

  Eine Anfrage stellt niemals eine vollständige und korrekte Spezifikation eines menschlichen Ziels dar. Der Softbot muß versuchen ein mögliches Ziel zu erkennen und zu verfolgen.

- AUSGEGLICHEN

  Der Softbot muß ein Gleichgewicht zwischen den Kosten für die Beschaffung von Informationen auf eigenem Weg und der erneuten Abfrage des Benutzers abschätzen können.

- INTEGRIERT

  Der Softbot beinhaltet ein aussagekräftiges und uniformes Interface zu einer großen Menge von Internet-Diensten und Hilfsprogrammen.

Der Softbot überprüft die Anfrage und plant sein weiteres Vorgehen. Folgende Fragen könnten dabei auftreten:

- Welches Dokument soll gesendet werden? (Wo ist es abgelegt?)
- Wie können die Memos versandt werden? (E-Mail, Fax ...?)
- Was soll passieren, wenn die Memos vertraulich sind?
- Was passiert, wenn die Zielperson nicht angetroffen wird? ...

Auch bei der Bearbeitung der Anfrage können Komplikationen auftreten:

- die Anfrage ist unvollständig
- Mehrdeutigkeit kann nicht vollständig ausgeschlossen werden
- die Anfrage kann mit den gegebenen Mitteln nicht ausgeführt werden

Für den Softbot ist es daher von größter Bedeutung, Eindeutigkeit zu erhalten. Er muß absolut sicher sein, daß er bei der Behandlung der Anfrage nur immer eine Möglichkeit besitzt, die ihn letztendlich zum Ziel führt. Natürlich ist nicht auszuschließen, daß er auf dem Weg dorthin immer wieder mehrere Möglichkeiten zur Verfügung stehen hat. Allerdings muß an dieser Stelle solange eine Überprüfung stattfinden bis bewiesen ist, daß der nun eingeschlagene Weg der einzig richtige ist. Der Softbot kann sich die hierfür notwendigen zusätzlichen Informationen auf zwei Wege beschaffen:

- eigene Suche im Internet
- erneute Abfrage des Benutzers

Von der zweiten Möglichkeit sollte allerdings eher abgesehen werden, da ständiges Rückfragen das Vertrauen beim Benutzer schwinden lassen könnte.

A.2.4 Softbot Planung

Um ein integriertes und zielorientiertes Interface zu konstruieren, wurden AI-Planungstechniken verwandt. Die Planungsfunktion des Softbots benutzt logische Ausdrücke, um die Ziele des Benutzers als Input zu beschreiben. Nachdem er in seinen Bibliotheken nach Aktionsschemen gesucht hat, die mögliche Informationsquellen, Datenbanken, Hilfsprogramme und Softwarebefehle beinhalten, generiert die Planungsfunktion dynamisch eine Sequenz, die dem Softbot helfen soll das gesteckte Ziel zu erreichen. Die Planungsfunktion besitzt die Fähigkeit, komplexe Aufgaben zu zerlegen und sie durch Divide-and-Conquer Techniken zu lösen. Wenn der Softbot das angestrebte Ziel nicht auf direktem Weg erreichen kann, versuch er zunächst 'Zwischenziele' zu erreichen, um über sie an das Hauptziel zu gelangen.

A.2.5 Ressourcen Integration

Die Planungsfunktion beruht auf dem Modell aller nutzbaren Internet-Ressourcen, die Antworten für die folgenden Fragen liefern können:

- Wie kann der Softbot die angebotenen Ressourcen sinnvoll nutzen?
- Welcher Effekt wird hierdurch erwartet?

Bei der Implementierung des Softbots muß darauf geachtet werden, daß auf neue Ressourcen flexibel reagiert werden kann und eine Nutzung durch den Softbot unproblematisch ist. Gerade in diesem Bereich ist es äußerst wichtig, die Lerntechniken so weiter zu entwickeln, daß ein solcher Vorgang automatisch ablaufen kann.

A.2.6 Unvollständige Spezifikation

Der Softbot akzeptiert unvollständige Spezifikationen und versucht, falls dies möglich ist, an die fehlenden Informationen heranzukommen. Es können drei Arten von Zielen spezifiziert werden.

- Grundziele

  Meldung, wenn sich eine bestimmte Person an einem speziellen Rechner eingelogt hat.
- existenziell quantifizierte Ziele

  Meldung, wenn sich eine bestimmte Person an irgendeinem Rechner eingelogt hat.
- erzwungene Ziele

  Meldung, wenn sich eine bestimmte Person an einem speziellen Rechner in einem bestimmten Institut eingelogt hat.

Durch Nachforschungen wurde herausgefunden, daß die erzwungenen Ziele am besten die Balance zwischen dem lästigen Nachfragen beim Benutzer und der Sofbot-Suche im Internet darstellen.

A.2.7 Softbot Sicherheit

Es wurde argumentiert, daß für ein integriertes und zielorientiertes Interface, intelligente Programme und Werkzeuge wie der Softbot zur Verfügung stehen müssen. Aber auch der Softbot benötigt ein paar Eigenschaften bezüglich der Sicherheit. Internet-Power-Tools wie der Softbot können auch Schäden anrichten, wenn sie unachtsam eingesetzt werden. Die größte Gefahr liegt in der Fähigkeit des Softbots, sich seinen eigenen Plan für die Durchführung seines Auftrags zusammenzustellen. Ein Sicherheitsmechanismus für einen Softbot sollte daher die folgenden Punkte beinhalten:

- SICHER

  Der Softbot sollte versuchen, destruktive Veränderungen zu unterlassen.

- ORDENTLICH

  Der Softbot sollte alles so hinterlassen, wie er es aufgefunden hat.

- SPARSAM

  Sorgsamer Umgang mit den vorhandenen Ressourcen.

- WACHSAM

  Der Softbot sollte Aktionen des Benutzers unterbinden, die unbeabsichtigte Konsequenzen haben könnten.

A.2.8 Quellen

Etzioni, O. and Weld, D. (1994). A softbot-based interface to the internet. Communications of the ACM (CACM), 37(7):72-76.

Etzioni, O. and Weld, D. (1998). A softbot-based interface to the internet. In Huhns, M. N. and Singh, M. P., editors, Readings in Agents, pages 77-81. Morgan Kaufmann, San Francisco, Calif.

A.3 Einführung in das Information–Retrieval

von André Masloch

Zusammenfassung

In der heutigen Zeit sind viele Dokumente in elektronischer Form verfügbar. Die Speicherung dieser Dokumente bereitet zwar keine Probleme, jedoch ist die Dokumentsammlung wertlos, wenn der Benutzer keine Möglichkeit hat, effizient auf Dokumente eines Themengebietes zuzugreifen. Hier zeigt sich die Aufgabe des *Information Retrieval*, das Methoden zum Strukturieren und Vergleichen von Dokumenten bereitstellt.

Dieser Artikel soll dem Leser einen Überblick über diese Methoden geben und geht am Ende näher auf das Vektormodell und Clustering ein.

A.3.1 Methoden des Information Retrieval

Dieser Abschnitt behandelt die verschiedenen Methoden, die im Information Retrieval benutzt werden, um Dokumentsammlungen zu strukturieren. Bei vielen dieser Techniken kann man Dinge wie *Stammwortreduktion* oder *Prefix-, Suffix- und Stoplisten* zur Vorbehandlung der Dokumente und Anfragen benutzen, um die Trefferquoten zu verbessern und die Geschwindigkeit zu steigern. Im folgenden werden die Techniken:

- Volltextsuche
- Signaturdateien
- Inversion
- Vektormodell und
- Latent Semantic Indexing

mit ihren jeweiligen Vor- und Nachteilen vorgestellt.

Volltextsuche

Bei dieser Technik wird (wie nicht anders zu erwarten) jedes Dokument in der Sammlung komplett nach den zu suchenden Schlüsselworten durchsucht. Vorteile sind hierbei die Vollständigkeit der Suche sowie der minimale Aufwand, um Dokumente zur Sammlung hinzuzufügen oder zu ändern. Außerdem ist kein Speicheroverhead vorhanden.

Allerdings hat die Volltextsuche auch einen (wesentlichen) Nachteil. Bei großen Daten-/Dokumentmengen (im GB-Bereich) kann die Antwortzeit der Suche sehr groß werden, da auf alle Daten zugegriffen werden muß. Aus diesem Grund wird diese Technik oft in Hardware realisiert, da diese sehr viel schneller als Software ist.

Signaturdateien

Bei dieser Technik wird für jedes Dokument eine Kennung (die sogenannte Signatur) abgespeichert, welche durch eine vom Entwickler vorgegebene Hashfunktion erzeugt wird. Anstatt der Dokumente werden dann die Signaturen miteinander verglichen. Es ist natürlich ebenso möglich anstatt ganzer Dokumente z.B. Absätze, Worte oder n-grams [Harrison, 1971] als kleinste Einheiten zu benutzen.

Die Vorteile von Signaturdateien sind einfache Implementierbarkeit und Parallelisierbarkeit. Neben schneller Einfügung sind auch Dinge wie Teilwortsuche oder Tolerierung von Tippfehlern möglich. Probleme können auftreten, wenn die Signaturdatei sehr groß ist. Neben der längeren Antwortzeit wird es dann auch schwieriger, Änderungen an schon aufgenommenen Dokumenten vorzunehmen.

Inversion

Die Inversion ist das „Arbeitspferd" in den verschiedenen Methoden des Information Retrieval. Auch bei sehr großen Datenmengen ist die Suche schnell beendet. Hier wird jedes Dokument durch eine Liste von Schlüsselworten repräsentiert. Der Trick besteht dann darin, diese Listen „umzukehren", so daß dann für jedes Schlüsselwort eine Dokumentliste vorhanden ist. Diese Listen werden dann in Indexdateien abgespeichert.Im Allgemeinen können auch mehrere Indexdateien existieren oder Indizes von Indizes erstellt werden.

Die Vorteile der Inversion sind (wie schon gesagt) die Geschwindigkeit sowie die sehr gute Unterstützung von Synonymen oder verschiedenen Wortformen (für jedes synonyme Wort wird dieselbe Liste benutzt). Desweiteren sind boolesche Kombinationen von Schlüsselworten effizient durchführbar, da z.B. für das logische Oder lediglich die Listen der Schlüsselworte zusammengefügt werden müssen. Allerdings bringt die Inversion auch Nachteile mit sich. Aufgrund der Datenstrukturen kann der Platzverbrauch bis zu 300% der Orginaldaten betragen. Die schnelle Suche wird mit dem großen Aufwand bei Einfügen eines neuen Dokumentes bezahlt. Außerdem kann bei den Schlüsselwortlisten eine Verzerrung auftreten, die die Suche verlangsamt oder verschlechtert (Es werden evtl. sehr viele Dokumente geliefert). Neuere Methoden versuchen diese Verzerrung zu vermeiden, indem kurze und lange Listen unterschiedlich behandelt werden und die Listen adaptiv wachsen. Das Speicherplatzproblem versucht man durch Komprimierung zu vermeiden.

Vektormodell

Die Dokumente werden hier als n-dimensionale Vektoren dargestellt, dessen Stellen jeweils ein Schlüsselwort repräsentieren. Zur Ermittlung der Stellenwerte wird ein binäres Gewicht oder eine andere Gewichtsfunktion verwendet.[1] Der Vergleich der Dokumente wird meist durch das Cosinusmaß (Skalarprodukt) realisiert.

Die Dimension der Vektoren wird anfangs vorgegeben. Man ermittelt die Schlüsselworte initial meist automatisch durch Indexing. Zur Reduktion der Ordnung werden dabei Stop-, Prefix- und Suffixlisten sowie Synonymwörterbücher benutzt.

Latent Semantic Indexing

LSI ist eine recht neue Technik, die große (konzeptionelle) Unterschiede zu den vorherigen Methoden aufweist, weil es versucht die Konzepte der Dokumente herauszufinden.

Die Dokumente werden als Vektoren von Termhäufigkeiten repräsentiert. Anschließend wird auf die Term-Dokument-Matrix das mathematisches Verfahren der „Singular Value Decomposition" angewandt und eine Dimensionsreduktion (z.B. von 10000 auf 200) durchgeführt. Dieses soll dazu führen die Ungenauigkeiten in den Dokumentbeschreibungen zu unterdrücken und die inhaltlichen Zusammenhänge zwischen den Dokumenten hervorzuheben.Näheres zu LSI ist in [Dumais et al., 1990] zu finden.

Nachteile von LSI sind die aufwendige anfängliche Berechnung der SVD und die nur schwer mögliche Hinzunahme von Dokumenten und Schlüsselworten ohne Neuberechnung der Matrix durch

[1] Im nächsten Abschnitt werden einige Gewichtsfunktionen vorgestellt.

die SVD und anschließende Reduktion. Allerdings bietet LSI im Gegensatz zum Clustering den Vorteil, die vorhandene Struktur herauszufinden. Beim Clustering wird die Struktur durch den Entwickler „vorgegeben".

A.3.2 Das Vektormodell

In diesem Abschnitt wird genauer auf das Vektormodell eingegangen. Es werden einige Gewichtsfunktionen vorgestellt und bewertet. Anschließend werden die Methode des Clustering, sowie zwei Methoden zur Clustererstellung vorgestellt.

Bewertungsfunktionen

Wie schon im Vektormodell beschrieben, wird entweder ein binäres Gewicht oder eine Gewichtsfunktion verwendet, um die Dokumente im Vektorraum zu plazieren. Da diese Plazierung über die Ähnlichkeit der Dokumente entscheidet, sollte die Gewichtsfunktion sorgfältig gewählt werden.

Termhäufigkeit - ocurrence frequency

$$\mathrm{FREQ}(\mathrm{Dok}, W)$$

Wie oft taucht das Wort W im Dokument Dok auf ? Einfach zu ermitteln und besser als binäres Gewicht.

Termspezifität - term specifity

$$\log(N) - \log(\mathrm{DOCFREQ}(W)) + 1$$

In wie vielen Dokumenten taucht W (nicht) auf ? Relativ einfach zu ermitteln und besser als binäres Gewicht.

tf-idf

$$\frac{\mathrm{FREQ}(\mathrm{Dok}, W)}{\mathrm{DOCFREQ}(W)}$$

Wie oft taucht W in Dok im Vergleich zu allen Dokumenten auf ? Besser als die vorherigen Gewichte, sonst ähnlich.

Optimales Gewicht

$$\mathrm{FREQ}(\mathrm{Dok}, W) * \underbrace{\frac{r_W * (I - i_W)}{(R - r_W) * i_W}}_{\mathrm{TERMREL}(W)^3}$$

Wie oft taucht W in Dok im Vergleich zu den relevanten und irrelevanten Dokumenten auf? TERMREL(W) gibt die Relevanz des Wortes W an. Die Ermittlung der Werte R, I, r_Wund i_W ist schwierig, da Relevanzaussagen schlecht durch den Computer gemacht werden können (Wäre es so, bräuchten wir ja die anderen Gewichte nicht :-)). Meistens ist also Expertenhilfe durch den Menschen nötig.

[3] R : Anzahl relevanter Dokumente, I : Anzahl irrelevanter Dokumente, r_W : Anzahl relevanter Dokumente mit Wort W, i_W : Anzahl irrelevanter Dokumente mit Wort W

Cluster

Clustering ist eine Methode im Vektorraummodell die versucht, ähnliche Dokumente in dieselben Gruppen (Cluster) eizuordnen. Im Normalfall wird dann für jede Gruppe ein Schwerpunktvektor bestimmt. Wird später ein Dokument bzw. eine Anfrage mit dem Cluster verglichen, kann anstatt der zahlreichen Vergleiche mit den Dokumenten im Cluster ersteinmal der Schwerpunktvektor des Clusters hinzugezogen werden. Dokumente können in mehreren Clustern auftauchen. Außerdem ist es möglich durch Clustering Hierarchien aufzustellen, d.h. Clustering von z.B. Clustern, Superclustern oder Hyperclustern.

Im Gegensatz zu LSI hat Clustering den Nachteil, daß vorhandene Strukturen nicht erkannt werden, sondern die Dokumente nach einem vorgegebenen Muster bzw. anhand einer empirischen Konstante in Cluster eingeteilt werden. Allerdings ist Clustering dafür auch wesentlich schneller als LSI und kann sehr viel leichter mit „dynamischen“ Umgebungen umgehen, da Einfügungen oder Änderungen besser unterstützt sind.

Clustererstellung

Für die Clustererstellung gibt es im wesentlichen 4 Kriterien, die ein Algorithmus erfüllen sollte :

1. **Stabilität bei Wachstum**, d.h. die Clusterstruktur ändert sich beim Einfügen eines neuen Dokumentes nicht gravierend.

2. **Robustheit**, d.h. kleine Fehler in den Dokumenten verursachen auch nur kleine Fehler in der Clusterstruktur.

3. **Anordnungsunabhängigkeit**, d.h. die Clusterstruktur hängt nicht davon ab, in welcher Reihenfolge der Algorithmus die Dokumente verarbeitet.

4. **Effizienz**, d.h. der Algorithmus sollte bei großen Dokumentmengen möglichst schnell sein (Platzbedarf wird i.A. vernachlässigt).

Leider gibt es bis jetzt keinen Algorithmus, der alle Kriterien erfüllt. Die auf der **Ähnlichkeitsmatrix** basierenden Verfahren erfüllen in den meisten Fällen die ersten drei Punkte, **iterative Verfahren** sind schnell.

Clustererstellung mit Ahnlichkeitsmatrix Zuerst wird eine Matrix aufgestellt, die die Ähnlichkeiten der Dokumente zueinander enthält. Anschließend kann die Clustereinteilung durch verschiedene Verfahren wie z.B.:

- Einteilung per Schwellwert. *Problem:* Wie Auswählen ?

- Graph und „single Link“-Kriterium. Ein hierarchisches Clustering mit Clustering durch Levelschwellwert [Rijsbergen, 1971]

vorgenommen werden.

Die Laufzeit der Algorithmen ist $O(N^2)$, da schon zum Aufbau der Matrix quadratische Zeit benötigt wird.

iterative Methoden zu Clustererstellung Die iterativen Methoden basieren darauf, mit einer initialen Partitionierung zu beginnen (wo auch immer diese herkommt). Anschließend wird diese durch wiederholtes Neuanordnen der Dokumente „verfeinert", bis keine Verbesserung mehr auftritt. Es leuchtet ein, daß diese Algorithmen stark davon abhängen, in welcher Reihenfolge die Dokumente bearbeitet werden. Dafür benötigen Verfahren dieser Art aber auch nur $O(N \log(N))$ Zeit. Eine Ausnahme scheint das „Single-Pass"-Verfahren [Salton and Wong, 1978] zu sein, daß in nur einem Durchlauf eine Clusterstruktur erzeugt[2]

Die Nachteile der iterativen Vefahren liegen in der hohen Anzahl von Parametern von denen die Clusterstruktur sehr stark abhängt, da man diese experimentell bestimmen muß. Der große Vorteil ist aber wie schon gesagt die Geschwindigkeit.

Suchen in Clustern - relevance Feedback

Hier gibt es eigentlich nicht mehr viel zu sagen. Jede Anfrage wird in einen Pseudo-Query-Vektor umgewandelt. Anschließend wird dieser mit den Schwerpunkten der einzelnen Cluster verglichen.

Eine Erweiterung bietet hier das „*relevance-feedback*". Durch Bewertung der Treffer vom Benutzer wird der ursprüngliche Query-Vektor verschoben. Dabei werden die Treffer vektoriell addiert und die anderen subtrahiert. Bereits nach wenigen (2-3) Iterationen erzielt man so erstaunlich gute Ergebnisse.

[2] Vorgehen : Jedes Dokument wird mit den bestehenden Clustern verglichen. Anhand eines vorgegebenen Schwellwertes wird entschieden, ob das Dokument in bestehende Cluster einsortiert wird oder einen neuen Cluster bildet.

A.4 Textklassifikation und maschinelles Lernen

von Ulla Mentel

Quellen:

„Text Categorization with Support Vector Machines: Learning with Many Relevant Features" von Thorsten Joachims [Joachims, 1997]
„Machine Learning" von Tom M. Mitchell [Mitchell, 1997]
„C 4.5 programs for machine learning" von J. Ross Quinlan [Quinlan, 1993]

A.4.1 Textklassifikation

Textklassifikation ist eine klassische Aufgabe des maschinellen Lernen. Es geht in den folgenden Beispielen um überwachtes Lernen, d.h. eine Beispielmenge (Trainingsmenge) wird klassifiziert, um Beispiele aus einer Testmenge einordnen zu können.
Wir gehen für *Support Vector Machines* und *k-Nearest Neighbors* von einer Textrepräsentation in der Form von Feature–Vektoren aus.

Merkmal (Feature) Auswahl

Es ist entscheidend, herauszufinden, welche Wörter einen Text charakterisieren, um ihn klassifizieren zu können. Es gibt verschiedene Methoden, z.B. :

DF-thresholding, d.h. Wörter, die sehr oft vorkommen (ist, hat, ein, usw.) und Wörter, die sehr selten vonkommen, werden nicht berücksichtigt, alle anderen zwischen diesen Grenzen werden als charakteristisch angenommen.

Information gain, man ermittelt den Informationsgewinn, das jedes Attribut (Wort) bringt und wählt das mit dem größten Gewinn.Der Informationsgehalt wird durch die Entropie angegeben:

$$-\sum_{i=1}^{n} p_i log_2 p_i \tag{A.1}$$

n :Attributwerte
p_i: Wahrscheinlichkeit für den i–ten Wert

Anforderungen an Klassifizierungsprogramme

Um herauszufinden, welche Methoden erfolgversprechend beim Lernen von Textklassifikation sind, müssen wir die Eigenschaften von Texten genau betrachten:

1. Hochdimensionale Eingabewerte

2. einige irrelevante Features (leider nur sehr wenige)

3. die Dokumentvektoren sind dürftig,(d.h. zu jedem Dokument d_i enthält der entsprechende Dokumentvektor $\overrightarrow{d}_i$ nur wenige Einträge, die nicht Null sind.)

A.4.2 Support Vektor Machines

[Joachims, 1997] basiert auf *Stuctural Risk Minimization*
Idee: eine Hypothese h zu finden, für die wir eine niedrige Fehlerwahrscheinlichkeit garantieren können
obere Schranke für eine Hypothese h:

$$P(error(h)) \leq train\_error(h) + 2\sqrt{\frac{d(ln\frac{2n}{d}+1) - ln\frac{\eta}{4}}{n}} \tag{A.2}$$

P(error(h)) : Fehlerwahrscheinlichkeit für Hypothese h

train_error(h) : Fehlerrate von h bei Trainingsmenge

n : Anzahl der Trainingsbeispiele

d : *VC-Dimension (VC-dim)*

VC-dim gibt die Komplexität des Hypotesenraums an und seine Größe steht im trade–off zur Fehlerrate bei der Trainingsmenge, d.h.:

VC-dim klein $\Leftrightarrow$ Fehlerrate groß
VC-dim groß $\Leftrightarrow$ Fehlerrate klein, aber „Überbewertung“ der features

Ziel: die „richtige“ Dimension
dafür definieren wir eine Schwellwertfunktion:

$$h(\vec{d}) = sign\{\vec{w}\cdot\vec{d}+b\} = \begin{cases} +1 & wenn \quad \vec{w}\cdot\vec{d}+b>0 \\ -1 & sonst \end{cases} \tag{A.3}$$

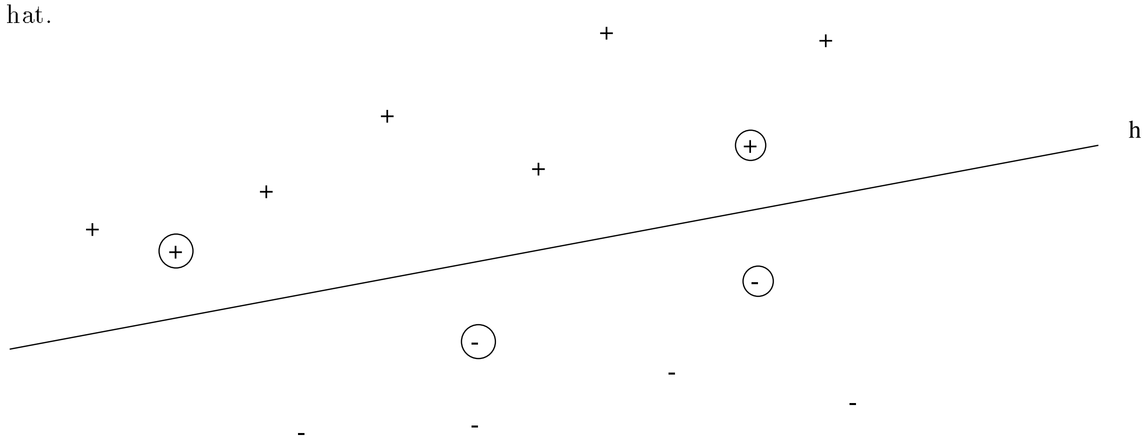
Lemma [Vapnik, 1982]Betrachte Hyperplanes$h(\vec{d}) = sign\{\vec{w}\cdot\vec{d}+b\}$ als Hypothesen. Wenn alle Beispielvektoren $\vec{d}_i$ in einer Kugel mit Radius R enthalten sind und für alle Beispiele $\vec{d}_i$ gilt

$$|\vec{w}\cdot\vec{d}+b| \geq 1, mit||\vec{w}|| = A \tag{A.4}$$

dann hat die Hyperplane die *VC-dim* d und ist begrenzt durch

$$d \leq min([R^2A^2], n) + 1 \tag{A.5}$$

VC-dim hängt von der Euklidischen Länge $||\vec{w}||$ des Gewichtsvektors $\vec{w}$ ab, d.h. wir können in hochdimensionalen Räumen klassifizieren, wenn unsere Hypothese einen kleinen Gewichtsvektor hat.



Surpport vector machines finden die Hyperplane, die die Beispiele aufteilt und den kürzesten Gewichtsvektor hat. Die Beispiele, die am nächsten an der Hyperplane liegen heißen *Support Vectors*.

A.4.3 k-Nearest Neighbors

Dieser Algorithmus setzt voraus, daß alle Beispiele einem Punkt in einem n- dimensionalen Raum $\Re^n$ entsprechen, das Beispiel x also durch den Vektor

$$(a_1(x), a_2(x), ..., a_n(x))$$

Wert des r–ten Attributs: $a_r(x)$

Abstand zwischen 2 Beispielen x_i, x_j:

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n}(a_r(x_i) - a_r(x_j))^2} \tag{A.6}$$

für $f : \Re^n \rightarrow \Re$ ist $\widehat{f}(x_q)$ eine Abschätzung

Grundalgorithmus:

Trainings- Algorithmus:

- Für jedes Trainingsbeispiel $(x, f(x))$, addiere dieses Beispiel zur Liste *trainings_ examples*

Klassifikations- Algorithmus:

- gegeben sei ein neues Beispiel x_q, daß klassifiziert werden soll
  - Seien $x_1, x_2, ..., x_k$ die k Beispiele aus *trainings_ examples*, die am nächsten zu x_q sind
  - return
$$\widehat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{A.7}$$

Nearest Neighbor mit gewichteter Distanz

Eine offendsichtliche Verbesserung ist es, den Beitrag von jedem der k Nachbarn gemäß seiner Distanz zum fraglichen Punkt x_q zu gewichten mit

$$w_i = \frac{1}{d(x_q, x_i)^2} \tag{A.8}$$

dann wird die letzte Zeile des Klassifikations–Algorithmus ersetzt durch

$$\widehat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{A.9}$$

wenn $f(x_i) = c$ für alle Beispiele gilt, dann ist $\widehat{f}(x_q) \leftarrow c$

A.4.4 Decision Tree Classifier

Quinlan's Ansatz ist ein „top–down“ Lernen aus Beispielen mit statistischer Merkmalselektion. Programme, die Entscheidungsbäume konstruieren

- an den Kanten stehen Attributwerte
- Knoten sind Verzweigungspunkte
- Blätter stellen Begriffsnamen (Klassen) dar

für ein neues Beispiel soll entschieden werden, zu welchem Begriff (welcher Klasse) es gehört: folge den Kanten, deren Beschriftung dem Attributwert des Beispiels entspricht, bis ein Blatt erreicht ist. Das Beispiel wird der Klasse zugeordnet, die an dem Blatt angegeben ist.

Die bekanntesten Realisierenden sind C 4.5, wie auch ID3, und haben ihren Ursprung in Hunts Concept Learning Systems. [Hunt et al., 1966]

Divide-and-conquer Algorithmus
(Hunt, 50-er Jahre)

Menge von Beispielen T, Klassen $\{C_1, C_2, ..., C_n\}$, es gibt 3 Möglichkeiten:

- T beinhaltet eine oder mehrere Fälle (Beispiele) und alle gehören zu einer einzelnen Klasse C
- T ist leer. Der Entscheidungsbaum ist dann ein Blatt, aber die Klasse,die mit diesem Blatt assoziiert wird, muß abgegrenzt sein gegenüber der Information anderer Mengen als T. [C4.5 benutzt die meisten frequentierte Klasse am Elternknoten dieses Knotens]
- T beinhaltet Fälle, die zu einer Mischung von Klassen gehört: T wird zu Untermengen verfeinert. Ein Test *TestX* wird ausgewählt, basierend auf einem einzelnen Attributs, das ein oder mehrere, sich gegenseitig ausschließende, Ergebnisse $\{O_1, O_2, .., O_n\}$ hat. T wird aufgeteilt in die Unterklassen $T_1, T_2, .., T_n$, wobei T_i alle Fälle aus T beinhaltet, die das Ergebnis O_i bzgl. des ausgewählten Tests haben. Der Entscheidungsbaum besteht aus einem, den Test identifizierenden Entscheidungsknoten und einem Zweig für jedes mögliche Ergebnis.
  Der selbe Baum-Bildungs–Mechanismus wird rekursiv aufgerufen.

Gewinn–Zuweisungs–Kriterium [C 4.5]

Forderung:
Der Baum soll aussagekräftig und möglichst klein sein.
Statt einer *Entropie* der Beispielmenge wird eine Art Normalisierung eingeführt, durch die der vermeintliche Gewinn zurechtgerückt wird.

Wir definieren

$$split\_info(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} * log_2 \frac{|T_i|}{|T|} \tag{A.10}$$

Dies repräsentiert den Informationsgehalt erzeugt durch die Teilung gemäß T in n Untermengen. Der Informationsgewinn wird durch die, für die Klassifizierung relevante Information bemessen, die sich aus derselben Teilung ergibt. So ist

$$gain\_info(X) = \frac{gain(X)}{splitinfo(X)} \tag{A.11}$$

Wenn die Teilung fast trivial ist, wird die split–Information klein und die Zuweisung unstabil sein.

Um dies zu vermeiden, wählt das Gewinn–Zuweisungs–Kriterium einen Test aus, der die Zuweisung maximiert.

Grund für die Einschränkung ist, daß der Informationsgewinn groß sein soll, mindestens so groß wie der durchschnittliche Gewinn über alle Testergebnisse.

Error- based- pruning

Die rekursive Aufteilungsmethode führt zu sehr komplexen Bäumen, weil die Daten „überbewertet“ werden. Es wird mehr Struktur gefolgert, als die Trainingsmenge rechtfertigt. C 4.5 erlaubt nach der Aufbauphase die Ersetzung von Unterbäumen durch Blätter oder durch einen seiner Zweige.Dadurch entstehen Fehler.

Angenommen es wäre möglich, die Fehlerrate eines Baumes, mitsamt seinen Unterbäumen und Blättern, vorherzusagen, dann wäre das Vorgehen:

- beginne unten am Ende des Baumes
- würde die Ersetzung eines Unterbaumes zu Reduzierung der Fehlerrate führen, dann reduziere entsprechend
- rufe den Prozeß rekursiv wieder auf, bis der Baum minimal ist

Wie kann die Fehlerrate vorhergesagt werden?

Die Wahrscheinlichkeit von Fehlern kann nicht genau festgestellt werden, hat aber selber eine Wahrscheinlichkeitsverteilung, die zusammengefaßt wird in einem Paar aus Sicherheisgrenzen.

Gegeben: ein Sicherheitslevel CF

> $\Rightarrow$ die untere Grenze wird berechnet durch die Binominalverteilung und wird geschrieben als U_{CF} (E,N) ,wobei N die Anzahl der Trainingsfälle ist, die durch ein Blatt überdeckt werden, E davon fälschlicherweise.

C 4.5 setzt diese untere Grenze mit der erwarten Fehlerrate gleich. Ein Blatt, das N Trainingsfälle überdeckt, mit einer angenommenen Fehlerrate von U_{CF}(E,N), würde einen Fehleranstieg von $N * U_{CF}(E, N)$ ergeben. Die Anzahl der vorausgesagten Fehler bezogen auf einen Baum ist genau die Summe der vorausgesagten Fehler an seinen Zweigen.
Es hat sich in Experimenten gezeigt, daß in dem reduzierten Baum die Fehlerrate bezogen auf die Testmenge wesendlich niedriger war als die der Trainingsmenge, vorallem dann, wenn die Testmenge sehr viel größer ist als die Trainingsmenge.

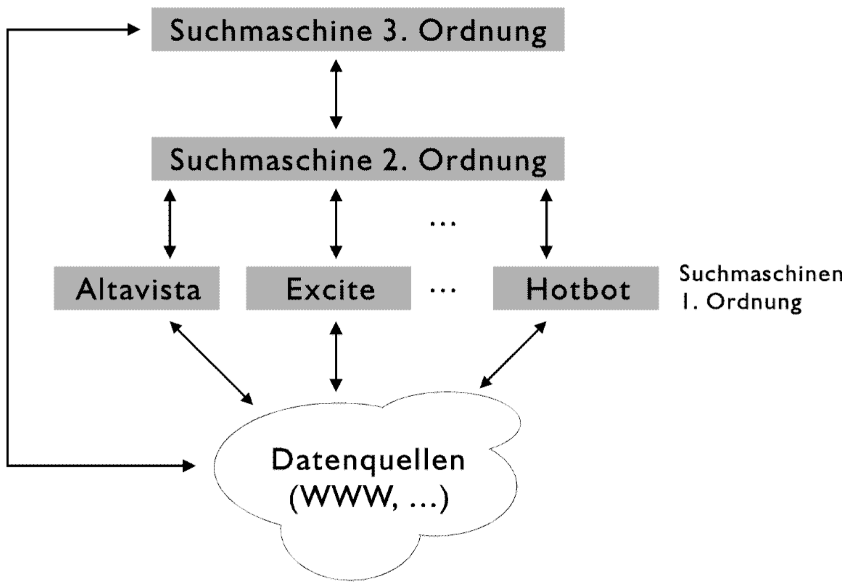
A.5 Ausgewählte Metasuchmaschinen

von Markus Hövener

A.5.1 Suchmaschinen

Ebenen-Einordnung

Ein Vorschlag für die Einordnungn von Suchmaschinen [Sander Beuermann, 1998] definiert Suchmaschinen erster, zweiter und dritter Ordnung.

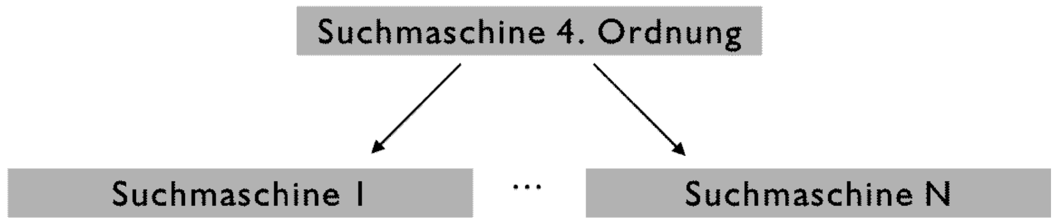


- Suchmaschinen erster Ordnung umfassen Suchdienste, die einen Index vollautomatisch erstellen (z.B. Altavista, Excite) sowei manuell erstellte Suchangebote (sog. Kataloge, z.B. Yahoo!).

- Suchmaschinen zweiter Ordnung fragen Suchergebnisse anderer Suchmaschinen (erster Ordnung) ab und fassen deren Ergebnisse zusammen.

- Suchmaschinen dritte Ordnungr nutzen die Ergebnisse von Suchmaschinen zweiter Ordnung, laden jedoch zusätzlich die gefundenen Dokumente herunter.

Es ist ebenfalls angedacht, Suchmaschinen vierter Ordnung zu etablieren. Diese nehmen eine Suchanfrage entgegen und fragen daraufhin Suchmaschinen ab, die auf das Problem spezialisiert sind bzw. voraussichtlich die besten Ergebnisse liefern (z.B. Jura-Datenbanken, medizinische Datenbanken).



Probleme der Suchmaschinen erster Ordnung

Das grundlegende Problem von Suchmaschinen erster Ordnung besteht in der Divergenz zwischen dem Index und der indizierten Information. Dokumente können gelöscht oder verändert werden, ohne daß die indizierende Suchmaschine dieses erfährt (z.B. dead links). Außerdem besteht das Problem des search engine spammings: durch Angabe von Meta-Tags werden Dokumente mit Stichworten verknüpft (Ziel: hohes Ranking).

Eine Studie [Lawrence and Giles, 1998b] belegt außerdem, daß keine Suchmaschine alle im Internet verfügbaren Dokumente indizieren kann (mangelnde Abdeckung). Bei der Untersuchung zugrundegelegt wurde eine geschätzte Anzahl von 320 Millionen im Internet verfügbaren Dokumenten. Dabei schneiden Hotbot mit 57,5% und AltaVista mit 46,5% noch sehr gut ab, während die Suchmaschinen InfoSeek mit 16,5% und Lycos mit 4,41% eher nicht zu empfehlen sind.

Ein weiteres Problem ist die fehlende Standardisierung der Suchmaschinen erster Ordnung. Die Unterschiede liegen vor allen im User-Interface, der Anfragesprache sowie der Features (Operatoren, Katalogfunktionen, etc.).

Meta-Suchmaschinen als Lösung des Problems

Das Problem der Divergenz zwischen Index und indizierter Informationen können Meta–Suchmaschinen nur bedingt lösen. Dazu ist eine Suchmaschine dritter Ordnung nötig, die die Ergebnisse der Suchmaschinen als Ausgangsbasis nimmt, aber dennoch alle gefundenen URLs überprüft.

Sinn einer Meta-Suchmaschine ist vor allem, die mangelnde Abdeckung einzelner Suchmaschinen dadurch zu umgehen, daß möglichst viele Suchmaschinen befragt werden und so eine sehr hohe, wenn auch nicht komplette Abdeckung erreicht wird.

Der Vorteil einer Meta-Suchmaschine ist vor allem in der Konzentration auf das Wesentliche zu sehen. Eine Anfrage an eine Meta-Suchmaschine sollte den User von der Fragestellung befreien, wo und wie er suchen will. Eine Meta-Suchmaschine stellt zu diesem Zweck und User-Interface bereit (single unified interface), das von den abzufragenden Suchmaschinen abstrahiert.

Anforderungen an Meta-Suchmaschinen

[Sander Beuermann, 1998] liefern einen Anhaltspunkt für den notwendigen Umfang von Meta–Suchmaschinen:

Parallele Suche: Die Meta-Suchmaschine muß parallel Anfragen an die Suchmaschine stellen können (keine All-In-One-Formulare).

Ergebnis-Merging: Die Ergebnisse müssen zusammengeführt und einheitlich dargestellt werden.

Doubletten-Eliminierung: Doppelte Fundstellen müssen erkannt und gekennzeichnet werden.

Vollständige Operatoren: Mindestanforderung: Bereistellung der Operatoren AND und OR.

Kein Informationsverlust: Wenn Suchmaschinen Kurzbeschreibungen liefern, müssen diese auch übernommen werden.

Hiding: Spezifika der einzelnen Suchmaschinen müssen vor dem Benutzer versteckt werden.

Vollständige Suche: Solange einer der Suchdienste noch Ergebnisse liefern kann, muß die Meta-Suchmaschine in der Lage sein, diese darzustellen.

Konkrete Meta-Suchmaschinen

Im folgenden werden Teilaspekte der folgenden Meta-Suchmaschinen behandelt:

MetaCrawler (zweite/dritte Ordnung): Department of Computer Science and Engineering, University of Washington (Selberg, Etzioni)

Inquirus (dritte Ordnung): NEC Research Institute (Lawrence, Giles)

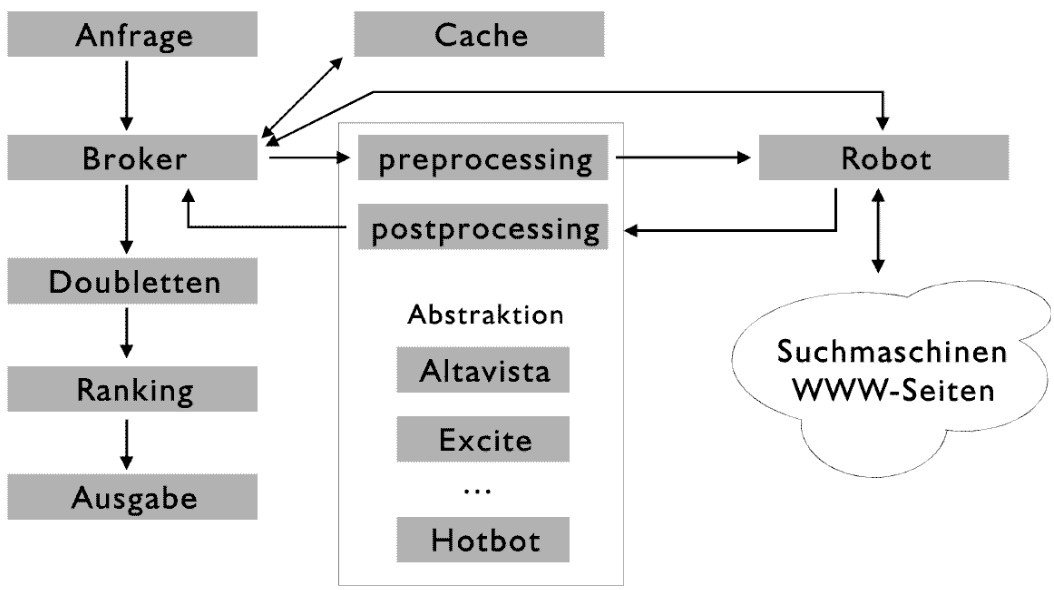
ProFusion (zweite Ordnung): Department of Electrical Engineering and Computer Science, University of Kansas (Gauch, Wang, Gomez)

MetaGer (zweite Ordnung): RRZN Hannover (Dr. Wolfgang Sander-Beuermann)

A.5.2 Komponenten von Meta-Suchmaschinen

Komponenten

Für eine Meta-Suchmaschine werden eine Vielzahl von Komponenten benötigt. Der Begriff Komponente ist in diesem Zusammenhang nicht unbedingt als eigenständiges Programm zu verstehen, auch wenn z.B. der Robot i.d.R. als Dämon/Service realisiert wird, über den der Broker via IPC kommuniziert.



Broker: CGI als Schnittstelle zwischen User und den Komponenten

Robot: Herunterladen von Dokumenten (GET-Requests), Überprüfen von Links (HEAD–Requests)

Cache: Speichern von Suchergebnissen

Preprocessing: Umwandlung der Suchanfragen in die Anfragesprache der jeweiligen Suchmaschinen

Postprocessing: Extraktion der Daten aus den HTML-Seiten (URL, Kurzbeschreibung, Ranking, Title)

Doubletten-Check: Dokumente gleichen Inhalts erkennen

Ranking: Wichtung der Seiten vornehmen (i.d.R. auf Basis des Rankings der abgefragten Suchmaschinen)

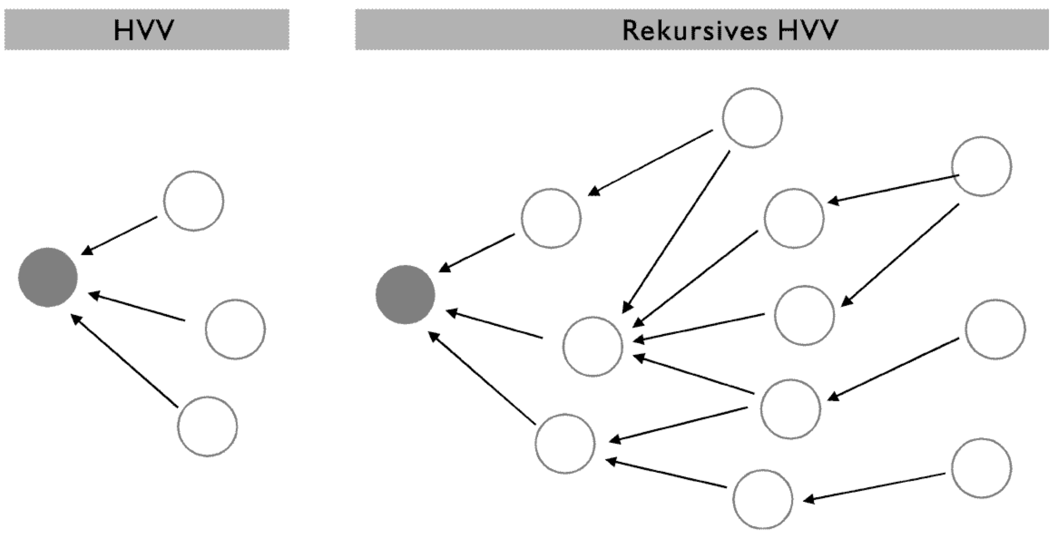
Exkurs: Ranking-Mechanismen

Die Ranking-Mechanismen (Bestimmung der Relevanz eines Dokumentes) der bekannten großen Suchmaschinen sind zu großen Teilen ein wohlgehütetes Geheimnis der Suchmaschinenbetreiber. Interessant in diesem Zusammenhang ist Dirk van Eylen's Analsye der Altavista-Ranking-Mechanismen [van Eylen, 1997].

Neben einfachen Mechanismen wie der unter 3.1 vorgestellte Algorithmus sollen vor allem zwei Methoden Erwähnung finden: CBR und HVV.

Kollaboratives Filtern (CBR) basiert auf der Rückmeldung der Suchenden, i.d.R. über das Verknüpfen der Suchanfrage mit den tatsächlich verfolgten Links (dies ist z. B. implementiert durch www.directhit.com). Dieses Verfahren ist allerdings leicht zu manipulieren, sei es durch Menschen oder Roboter.

Hyperlink Vector Voting (HVV) geht von folgender Annahme aus: Je mehr Links auf ein Dokument zeigen, desto relevanter/besser ist es. Da auch dieses Verfahren relativ leicht manipulierbar ist, wurde HVV um den Gedanken der Rekursivität erweitert: Die Links, die auf ein Dokument zeigen, werden gewichtet durch die Links, die auf die darauf zeigende Site zeigen, etc. Implementiert ist dieses Verfahren in der Suchmaschine Google! (google.stanford.edu).



Beide Verfahren können jedoch bei Meta-Suchmaschinen keine Verwendung finden. Für solche Suchmaschinen bleiben nur zwei Möglichkeiten: entweder nutzen sie das Ranking der abgefragten

Suchmaschinen oder ranken nach einer Methode, die auf ein einzelnes Dokument anwendbar ist (siehe 3.1).

A.5.3 Implementierungsdetails an konkreten Beispielen

Im folgenden beschreibe ich einige konkrete Lösungen, die von den unterschiedlichen Suchmaschinenbetreibern entwickelt und eingesetzt werden.

Ranking bei Inquirus

Inquirus nutzt das Ranking der abgefragten Suchmaschinen nicht. Um dennoch die Relevanz der Fundstellen beurteilen zu können, werden nach dem Befragen der Suchmaschinen die gefundenen Dokumente heruntergeladen und nach einer einheitlichen Formel gerankt:

$$c_1 \cdot N_p + (c_2 - \frac{\sum_{i=1}^{N_p-1} \sum_{j=i+1}^{N_p} min(d(i,j), c_2)}{\sum_{k=1}^{N_p-1} (N_p - k)}) \cdot \frac{c_1}{c_2} + \frac{N_t}{c_3}$$

In die Brechnung der Relevanz eines Dokumentes fließen folgende Variablen ein:

N_p: Anzahl der Suchbegriffe, die im Dokument vorkommen (einfach gezählt)

N_t: Anzahl der Suchbegriffe, die im Dokument vorkommen (mehrfach gezählt)

d(i,j): Abstand zwischen dem i-ten und dem j-ten Suchbegriff, der im Dokument vorkommt

c_1: Konstante, die R kontrolliert (z.Z. 100)

c_2: Konstante, die den Maximalabstand zweier Begriffe kontrolliert (z.Z. 5000)

c_3: Konstante, die die Wichtigkeit der Begriffshäufigkeit kontrolliert (z.Z. 10 c_1)

Context-Highlighting bei Inquirus

Die von Suchmaschinen gelieferten Kurzbeschreibunge sind bis auf wenige Ausnahmen (z.B. crawler.de) statisch, i.d.R. spiegeln sie den Anfang eines Dokumentes oder die über Meta-Tags gesetzte Beschreibung wider. Inquirus analysiert im Gegensatz dazu das gesamte Dokument und zeigt den die Suchbegriffe umgebenden Kontext an.

Specific expressive Forms (SEF) bei Inquirus

Inquirus erlaubt in gewissem Umfang umgangssprachliche Aussagen, z.B. „What does X stand for?“ oder „What causes X?“. Hierzu werden die Anfragen konvertiert, z.B. „What does X stand for?“ nach „X stands for“ OR „X is an abbreviation“ OR „X means“.

Doubletten-Filterung bei ProFusion

Um Doubletten herauszufiltern, vergleicht ProFusion alle Suchergebnisse miteinander. Der Algorithmus geht dabei mehrschrittig vor:

Überprüfung der URLs auf absolute Übereinstimmung Beachtung diverser Regeln für URLs (http://server/ = http://server/index.html) Falls die Titel zweier Fundstellen identisch sind, werden die Pfade der URLs auf Ähnlichkeit überprüft. Potentiell doppelte URLs könnten heruntergeladen werden und dann Übereinstimmung überprüft werden. Dieses Feature ist zwar implementiert, aber nicht öffentlich zugänglich gemacht worden.

Merging bei ProFusion

Das Zusammenfassen der Suchergebnisse basiert auf dem Ranking der einzelnen abgefragten Suchmaschinen. Dazu wurden für jede Suchmaschine sogenannte Confidence Factor CF_i ermittelt, die von der Qualität der Suchmaschinen abhängen (ermittelt auf der Basis von 25 verschiedenen Anfragen). Die Confidence Factors der von ProFusion befragten Suchmaschinen liegen zwischen 0,75 und 0,85.

Jedes Dokument besitzt einen Match Factor M_{di} zwischen 0 und 1 (den Wert 1 erhält das Dokument mit dem höchsten Ranking). Danach wird das Relevance Weight W_{di} als Produkt aus M_{di} und CF_i berechnet. Falls anschliessend zwei Dokumente bei der Doubletten-Filterung verschmelzen, erhält das übriggebliebene Dokument der Maximum der W_{di}.

Automatische Auswahl von Suchmaschinen bei ProFusion

ProFusion hat 13 Kategorien (z.B. „Science and Engineering“, „Travel“ und „Computer Science“) festgelegt und zu jeder dieser Kategorien Faktoren ermittelt, die besagen, wie relevant die Ergebnisse einer Suchmaschine zu Suchanfragen dieser Kategorie sind (Maximum: 0,767, Altavista, „Food“; Minimum: 0.011, OpenText, „History“).

Auf Basis dieser Faktoren wählt ProFusion auf Wunsch drei „optimale“ Suchmaschinen aus. Problematisch hierbei: eine Suchanfrage kann in verschiedene Kategorien passen, ein einzelnes Suchwort ebenso. Der Algorithmus beginnt damit, jeder Suchmaschine einen auf null initialisierten Zähler zuzuweisen. Danach werden für alle Suchworte die Kategorien ermitteln. Die Faktoren dieser Kategorien werden dann den jeweiligen Zählern aufaddiert. Ausgewählt werden dann die drei Suchmaschinen mit den höchsten Zählern.

QuickTips bei MetaGer

Um die Wartezeit während der Anfragen an die Suchmaschinen zu verkürzen, durchsucht MetaGer parallel eine Liste mit deutschen und internationalen Domainnamen (.de und .com), in denen nach den Suchbegriffen gesucht wird. Eine Suche nach „Uni dortmund“ würde z.B. Links auf www.dortmund.de und www.uni-dortmund.de liefern. Neben der Nützlichkeit dieser Information steht vor allem der Wunsch im Vordergrund, die Wartezeit geringer erscheinen zu lassen. Für die Zukunft ist an den Anschluß weiterer lokaler Datenbanken gedacht, z.B. an die USENET-Datenbank des RRZN Hannover.

Vervollständigung der Operatoren beim MetaCrawler

Einige Funktionen werden nicht von allen Suchmaschinen unterstützt, z.B. bietet Lycos keine Phrasensuche. Um dieses Feature dennoch anzubieten, wird bei fehlender Phrasensuche zuerst eine AND-Anfrage durchgeführt, um danach die gefundenen Dokumente per Robot herunterzuladen und auf diese Phrase zu überprüfen.

Erweiterte Optionen beim MetaCrawler

MetaCrawler bietet u.a. die Einschränkung auf Hosts ein, entweder geographisch (z.B. Europa) oder nach Domains (z.B. .de).

Optimierte Ausgabeformate

Alle Meta-Suchmaschinen müssen zur Lösung ihrer Aufgabe HTML-Seiten parsen. Es erscheint allerdings wenig sinnvoll, zuerst HTML-Seiten zu generieren, diese über das Netz zu transferieren und danach zu parsen; sinnvoller wäre es in Hinsicht auf Netzlast und Rechenzeit, nur die wirklich relevanten Teile zu übertragen.

Zu diesem Zweck gibt es z.B. seitens der Suchmaschinen Eule und Nathan ein (nicht zueinander kompatibles) Ausgabeformat, das lediglich URLs, Kurzbeschreibungen und Titel der Fundstellen in einer Text-Datei überträgt.

A.5.4 Zukunftsideen und Perspektiven

Die vier untersuchten Meta-Suchmaschinen haben für die Zukunft folgende Ideen offengelegt:

- Inquirus
  - Klassifikation von Seiten
  - Verbesserte SEF
- ProFusion
  - Beurteilung der Seiten (viel Text, viel Links, ...)
  - Suchanfragen automatisch periodisch wiederholen (nur Änderungen übermitteln)
- MetaGer
  - Verständnis für Suchanfragen: Rückfragen bei mehrdeutigen Anfragen
- MetaCrawler
  - Bei gesteigerter Anzahl von Suchmaschinen kontext-basierte Auswahl der besten Suchmaschinen

Neben den bekannten Allround-Suchmaschinen entwickeln sich auch zunehmend Meta-Suchmaschinen für sehr abgegrenzte Bereiche, z.B. QualitySearch (Meta-Suche in Archiven großer Zeitungen und Zeitschriften), YourPosition (Ermitteln der Position der eigenen Homepage bei allen großen Suchmaschinen) oder NewsMeta (Sammeln von aktuellen Nachrichten aller großen Zeitungen und Zeitschriften und Einsortieren in Themenbereiche).

A.5.5 Literatur

Literatur zu diesem Thema sind u. a. [Lawrence and Giles, 1998a], [Selberg and Etzioni, 1995], [Selberg and Etzioni, 1997] und [Susan Gauch, 1996] .

A.6 Internetprogrammierung

von Marina Podvoiskaia

A.6.1 HTTP

Das Protokoll, das für Transaktionen im WWW verwendet wird, ist das Hypertext Transfer Protocol (HTTP). Die Stöberer sind technisch gesehen HTTP-Klienten, die an HTTP-Dienstrechnern (Web-Server) Anfragen stellen. Der Server stellt auf die entsprechende Anfrage das gewünschte Dokument zur Verfügung.

Alle HTTP-Transaktionen folgen demselben Grundmuster. Jede Anforderung des Klienten und jede Antwort des Dienstrechners besteht aus drei Teilen:

1. Anforderungs- oder Antwortzeile
2. Kopfteil
3. Inhalt

Bei einer Anforderung eines Klienten enthält die erste Zeile einer Nachricht immer einen HTTP-Befehl (die sogenannte Methode), einen URL, der die angeforderte Datei benennt, und die HTTP-Versionsnummer. Diefolgenden Kopfzeilen enthalten Informationenüber den Klienten und die an den Dienstrechner gesendeten Daten. Der Inhalt ist meistens leer, es sei denn man verwendet die POST-Methode.

Die Antwort eines Dienstrechners sieht so aus: In der ersten Zeile ist die HTTP-Versionsnummer, eine Nummer, die den Zustand der Anforderung bestimmt, und eine kurze Zustandsbeschreibung enthalten. In den folgenden Kopfzeilen sind Informationen über den Dienstrechner und Informationen über das nun im Inhalt folgende Dokument enthalten.

- GET
  In einem HTML-Formular erzwingen Sie diesen Befehl durch die Angabe von method=get im einleitenden <form>-Tag. Bei dieser Angabe werden die ausgefüllten Formulardaten zuerst an die Server-Software übertragen und von dieser in einer bestimmten Umgebungsvariablen mit dem Namen QUERY_STRING zwischenspeichert. Ein CGI-Script, das durch die Angabe action= im einleitenden Formular-Tag aufgerufen wird, muß den Inhalt dieser Umgebungsvariablen auslesen, um an die Formulardaten heranzukommen. Wenn ein HTML-Formular die GET-Methode verwendet, wird der Formulardatenstrom, getrennt durch ein Fragezeichen, direkt hinter die URL-Adresse des CGI-Programmaufrufs gehängt. Im WWW-Browser des Anwenders ist dies nach dem Absenden des Formulars in der URL-Zeile sichtbar.
- POST
  In einem HTML-Formular erzwingen Sie diesen Befehl durch die Angabe von method=post im einleitenden <form>-Tag. Die Angabe bewirkt, daß die ausgefüllten Formulardaten direkt an die Adresse übertragen werden, die bei action= angegeben ist. Ein CGI-Script, das bei action= aufgerufen wird, muß die Standardeingabe auslesen, um an die Formulardaten heranzukommen.

A.6.2 HTML

Allgemein

HTML bedeutet HyperText Markup Language. Es handelt sich dabei um eine Sprache, die mit Hilfe von SGML (Standard Generalized Markup Language) definiert wird. SGML ist als ISO-Norm 8879 festgeschrieben.

HTML ist eine sogenannte Auszeichnungssprache (Markup Language). Sie hat die Aufgabe, die logischen Bestandteile eines Dokuments zu beschreiben. Als Auszeichnungssprache enthält HTML daher Befehle zum Markieren typischer Elemente eines Dokuments, wie Überschriften, Textabsätze, Listen, Tabellen oder Grafikreferenzen.

Das Beschreibungsschema von HTML geht von einer hierarchischen Gliederung aus. HTML beschreibt Dokumente. WWW-Browser, die HTML-Dateien am Bildschirm anzeigen, lösen die Auszeichnungsbefehle auf und stellen die Elemente dann in optisch gut erkennbarer Form am Bildschirm dar.

Schema des Grundgerüsts einer HTML-Datei

Eine gewöhnliche HTML-Datei besteht grundsätzlich aus folgenden zwei Teilen:

- Header (Kopf) (enthält Angaben zu Titel u.ä.)
- Body (Körper) (enthält den eigentlichen Text mit Überschriften, Verweisen, Grafikreferenzen usw.)

```
<html>
       <head>
              <title>Titel der Datei</title>
       </head>
       <body bgcolor=#XXXXXX text=#XXXXXX link=#XXXXXX
                        vlink=#XXXXXX alink=#XXXXXX
                    Inhalt der Datei
                <!-- Dies ist ein Kommentar -->
       </body>
</html>
```

Tabellen

Eine Tabelle besteht aus mindestens einer, normalerweise aus mehreren Zeilen. Eine Zeile besteht aus mindestens einer, normalerweise aus mehreren Zellen. Dadurch ergeben sich die Spalten der Tabelle.

```
<table border=x cellspacing=x cellpadding=x>
        <tr>
               <th>Kopfzelle</th>
               <th>Kopfzelle</th>
        </tr>
        <tr>
               <td>Datenzelle</td>
               <td>Datenzelle</td>
        </tr>
 </table>
```

Formulare

HTML stellt die Möglichkeit zur Verfügung, mit Hilfe spezieller Befehle Formulare zu erstellen. In Formularen kann der Anwender Eingabefelder ausfüllen, in mehrzeiligen Textfeldern Text eingeben, aus Listen Einträge auswählen und Buttons anklicken. Wenn das Formular fertig ausgefüllt ist, kann der Anwender auf einen Button klicken, um das Formular abzusenden.

Formulare können sehr unterschiedliche Aufgaben haben. So werden sie zum Beispiel eingesetzt:

- um bestimmte, gleichartig strukturierte Auskünfte von Anwendern einzuholen,
- um Anwendern das Suchen in Datenbanken zu ermöglichen,
- um Anwendern die Möglichkeit zu geben, selbst Daten für eine Datenbank beizusteuern,
- um dem Anwender die Möglichkeit individueller Interaktion zu bieten, etwa um aus einer Produktpalette etwas Bestimmtes zu bestellen.

Verweise

Eine der wichtigsten Eigenschaften von HTML ist die Verweise zu definieren. Verweise („Hyperlinks") können zu anderen Stellen im eigenen Projekt führen, aber auch zu beliebigen anderen Adressen im World Wide Web und sogar zu Internet-Adressen, die nicht Teil des WWW sind.

A.6.3 XML

Am 10. Februar wurde die Version 1.0 der XML-Spezifikation zu einem offiziellen Standard.

XML ist noch längst keine ausgereifte Technologie. Zwar kann man s agen, daß die Spezifikation festlegt, wie XML funktioniert. Sicher ist aber auch, daß es sehr viele Einzelfragen gibt, die erst geklärt werden, wenn an konkreten Projekten gearbeitet wird.

XML ist eine Metasprache für das Definieren von Dokumenttypen. Anders gesagt: XML liefert die Regeln, die beim Definieren von Dokumenttypen angewendet werden.

Und was ist ein Dokumenttyp? Klären wir zunächst die etwas einfachere Frage, was unter einem Elementtyp zu verstehen ist. Man sagt, daß dies ein Element ist: <p>Lola rennt.</p> (Das Element besteht aus einem Start-Tag, dem Inhalt des Elements und einem End-Tag.)

Auch dies ist ein Element: <p>Ist ja gar nicht wahr.</p> Es handelt sich um zwei Elemente mit auffallenden Ähnlichkeiten. Der Start-Tag und der End-Tag sind gleich. Deswegen sagt man, daß beide Elemente demselben Elementtypangehören.

Mit Dokumenten verhält es sich ähnlich. Wenn in Dokumenten dieselben Elementtypen verwendet werden (p und ul und br und was es da alles geben mag) und wenn die Elemente alle in gleicher Weise ineinander verschachtelt sind, dann sagt man, daß die Dokumente demselben Dokumenttyp angehören.

Wie steht es in dieser Hinsicht mit HTML-Dokumenten? HTML-Dokumente sind Musterbeispiele für Dokumente, die einen ähnlichen Aufbau haben. Man sagt daher, daß alle HTML-Dokumente demselben Typ angehören, nämlich dem Dokumenttyp HTML.

Letztlich ist das die Grundidee von XML: Man sorgt dafür, daß es Dokumente gibt, die alle in ihrem Aufbau gewissen Grundmustern folgen. Wenn diese Grundmuster eingehalten werden, dann läßt sich mit den Dokumenten mehr anfangen, als wenn jedes Dokument eigenen Regeln folgt, denn es ist dann möglich, Programme zu schreiben, die die Dokumente automatisch verarbeiten. Die Programmierer wissen in etwa, welche Strukturen in dem Dokument zu erwarten sind, und sie können Programme schreiben, die mit den Dokumenten umgehen können. Das heißt, die Dokumente werden keine Überraschungen mehr bieten, und die Programme können so eingerichtet werden, daß sie bei der Verarbeitung der Dokumente in jeder Situation wissen, was zu tun ist.

Dokumenttyp-Definitionen (DTDs) spielen in XML eine wichtige Rolle. In einer DTD wird festgelegt, welche Gemeinsamkeiten die Dokumente aufweisen werden. Zum Beispiel wird festgelegt, welche Elementtypen in den Dokumenten eines bestimmten Dokumenttyps verwendet werden können.

DTDs haben diese beiden Hauptfunktionen: Sie sagen den Verfassern von Dokumenten, welche Strukturen es in den Dokumenten geben kann. Und sie sagen den Programmierern, auf was ihre Programme sich gefaßt machen müssen.

Was ist also XML? In erster Linie ist XML eine Anleitung für das Verfassen von Dokumenttyp-Definitionen. Etwas Neues sind Dokumenttyp-Definitionen nicht. Es gab sie auch schon für HTML, man hat sich nur bisher meistens nicht sonderlich darum gekümmert, daß es da sowas im Hintergrund gab.

A.6.4 PERL/CGI

Allgemeines zu Perl

Perl steht für Pratical Extraction and Report Language. Die Sprache stammt aus der Unix-Welt und erblickte 1987 das Licht der Welt.

Ab der Version 5.0, unterstützt Perl auch den Ansatz der objektorientierten Programmierung. Jedoch ist es eine Script-Sprache, deren Haupteinsatz nicht umfangreiche Anwendungen sind, sondern trickreiche Automatismen in der täglichen Datenverarbeitung. Einen wahren Boom erlebt die Sprache aber vor allem als Lieblingswerkzeug der CGI-Programmierer im World Wide Web.

Perl-Dateien sind einfache ASCII-Dateien, die Programmanweisungen in der Syntax von Perl enthalten. Solche Dateien können Sie mit jedem Texteditor erstellen und bearbeiten. Zum Ausführen von Dateien mit Perl-Programmanweisungen ist jedoch der Perl-Interpreter erforderlich.

Skalare - einfache Variablen

Variablen sind Speicherbereiche, in denen Sie Daten, die Sie im Laufe Ihrer Programmprozeduren benötigen, speichern können. Der Inhalt, der in einer Variablen gespeichert ist, wird als Wert bezeichnet. Sie können den Wert einer Variablen jederzeit ändern.

In Perl wird eine einfache Variable, die eine Zahl oder eine Zeichenkette speichern kann, als Skalar bezeichnet.

Beispiele:

```
$Name = "Methusalem";
$Alter = 625;
$Name_2 = "Junger Hansel";
$JungAlter = sqrt($Alter);
```

Daten in Perl sind nicht „getypt". Wenn Sie eine Zeichenkette haben, die nur aus gültigen numerischen Zeichen besteht (z.B. „7423.13"), können Sie damit problemlos numerische Operationen durchführen. Ebenso können Sie numerische Werte wie Zeichenketten behandeln.

Listen - Arrays

Listen sind Ketten zusammengehöriger Skalare. In Perl haben solche Listen zentrale Bedeutung, weil es sehr einfach ist, Daten in Listen einzulesen und Listen zu manipulieren.

Beispiel 1:

```
#!/usr/bin/perl
@Daten = ("Jana",23,"Berlin","Abitur");
print $Daten[0], " ist ", $Daten[1], " Jahre alt, wohnt in ",
                            $Daten[2], " und hat ", $Daten[3];
```

Beispiel 2:

```
#!/usr/bin/perl
for($i = 1; $i <= 9; $i++)
 {
     $Wert = $i * $i;
     push(@Quadrate, $Wert);
 }
for(@Quadrate)
 {
     print $_;
 }
```

Reguläre Ausdrücke

Reguläre Ausdrücke sind genau definierte Suchmuster für Zeichen und Zeichenketten. Mit Hilfe dieser Suchmuster können Sie beispielsweise Variableninhalte durchsuchen und bearbeiten. Für CGI-Aufgaben sind reguläre Ausdrücke enorm wichtig. Zum Beispiel läßt sich der Datenstrom eines HTML-Formulars, der an ein CGI-Programm beim Absenden des Formulars übergeben wird, mit Hilfe von regulären Ausdrücken in seine einzelnen Bestandteile zerlegen. Genauso können Sie mit Hilfe von regulären Ausdrücken beim Einlesen von Dateien (z.B. einer Datei mit Einträgen eines Gästebuchs) anhand der Konventionen, nach denen die Datei aufgebaut ist, die einzelnen Einträge geordnet einlesen und als HTML-Code an den aufrufenden WWW-Browser übertragen. Und schließlich sind reguläre Ausdrücke ein mächtiges Mittel, um große Datenbestände nach komplexen Suchausdrücken zu durchforsten.

```
#!/usr/bin/perl

if ($Daten =~ /PERL/)    # Ob darin wohl die Zeichenfolge
                                  'PERL' vorkommt?
else ...

#!/usr/bin/perl
@Orte=("Berlin","Paris","London","Madrid","Athen","Rom",
                  "Lissabon","Stockholm", "Kopenhagen");

for(@Orte)
   {
      if(/[MKS]/)   # Alle Orte, in denen 'M', 'K' oder
                    #'S' vorkommt
      {print "Die Stadt ", $_, " entspricht dem Suchmuster";}
   }
```

Beispiele:

```
/a/            # findet 'a'
/[a-c]/        # findet 'a' oder 'b' oder 'c'
/\d/           # findet Ziffern
/\D/           # findet alles außer Ziffern
/[0-9]\-/      # findet Ziffern oder Minuszeichen
/[\w]/         # findet Buchstaben, Ziffern oder Unterstrich
/[\r]/         # findet das Steuerzeichen für Wagenrücklauf
/[\s]/         # findet Leerzeichen sowie Steuerzeichen
```

```
/[^a-zA-Z]/   # findet alles, worin keine Buchstaben vorkommen

/aus/         # findet 'aus' - auch in 'Haus' oder 'Mausi'
/Ha.s/        # findet 'Haus' und 'Hans' aber nicht 'Hannes'
/Ha.+s/       # findet 'Haus' und 'Hans' und 'Hannes'
/x{10,20}/    # findet zwischen 10 und 20 'x' in Folge
```

CGI-Schnittstelle

CGI (Common Gateway Interface) erlaubt es einem WWW-Browser, über einen WWW-Server Programme auszuführen. Solche Programme (oder Scripts) können beispielsweise Formulareingaben aus HTML-Dateien verarbeiten, auf dem Server-Rechner Daten speichern und dort gespeicherte Daten auslesen. Auf diese Weise werden WWW-Seiten zu Oberflächen für „Anwendungen", beispielsweise für elektronische Warenbestellung oder zum Abfragen von Datenbanken.

Die CGI-Schnittstelle besteht aus:

- einem bestimmten Verzeichnis auf dem Server-Rechner, das CGI-Programme enthalten darf. Meist erhält dieses Verzeichnis den Namen cgi-bin oder cgi-local. CGI-Programme oder CGI-Scripts werden dann nur ausgefüht, wenn sie in diesem Verzeichnis liegen.
- einer Reihe von Daten, die der WWW-Server speichert, und die ein CGI-Script auslesen kann (und zum Teil auslesen muß), um Daten verarbeiten zu können. Diese Daten speichert der WWW-Server in sogenannten CGI-Umgebungsvariablen.

HTML und CGI kommunizieren in beide Richtungen: es ist einerseits möglich, aus einer HTML-Datei, die gerade am Bildschirm angezeigt wird, ein CGI-Script aufzurufen; anderseits kann ein CGI-Script HTML-Code an den WWW-Browser übertragen, den dieser dann am Bildschirm ausgibt.

Ein CGI-Script kann Daten verarbeiten, die von einer HTML-Datei aus beim Aufruf übergeben werden. Zum Beispiel kann ein CGI-Script eine Datenbank durchsuchen, wobei der Anwender den Begriff, nach dem gesucht werden soll, in einem Formular angegeben hat. Die Ergebnisse einer Datenverarbeitung kann ein CGI-Script an den WWW-Browser in Form von HTML-Code zurücksenden. So kann ein Script, das eine Datenbank nach Begriffen durchsucht, zum Beispiel die Suchtreffer eines Suchvorgangs in Form einer dynamisch generierten HTML-Datei an den WWW-Browser zurücksenden.

CGI-Scripts können auch Daten auf dem Server speichern und zu einem späteren Zeitpunkt auslesen. Auf diesem Prinzip basieren beispielsweise Gästebücher oder Nachrichtenforen (Bulletin-Boards). Ein Anwender kann in einer HTML-Datei in einem Formular einen Beitrag eingeben. Beim Absenden des Formulars wird ein CGI-Script aufgerufen, das den Beitrag in einer Datei speichert. Ein zweites CGI-Script oder ein anderer Aufruf des CGI-Scripts kann anschließend HTML-Code mit allen gespeicherten Beiträgen an einen WWW-Browser übertragen.

Das Beispiel bewirkt einen einfachen CGI-Vorgang, der die Wechselwirkung zwischen HTML und CGI verdeutlicht: in einer HTML-Datei kann der Anwender in einem Formular seinen Namen und einen Kommentartext eingeben. Wenn er das Formular absendet, wird ein CGI-Programm comments.pl aufgerufen. Dieses Script ist in Perl geschrieben. Es liest die ankommenden Formulardaten ein, splittet sie in ihre Bestandteile auf und erzeugt eine vollständige HTML-Datei, in der es die eingelesenen Daten ausgibt. Der WWW-Browser zeigt diesen von comments.pl generierten HTML-Code am Bildschirm an.

```
<html>
    <head>
```

```
        <title>Kommentarseite</title>
    </head>
    <body>
        <h1>Ihr Kommentar</h1>
        <form action="/cgi-bin/comments.pl" method=post>
              Name:   <input name="AnwenderName" size=40>
              E-Mail: <input name="AnwenderMail" size=40>
              Text:   <textarea name="Text"></textarea>
          <input type=submit value="Formulardaten absenden">
        </form>
    /body>
</html>
```

Datenstrom bei Übertragung von Formulardaten

Ein typisches HTML-Formular besteht aus benannten Feldern (z.B. für Name, E-Mail-Adresse und Kommentartext). Bei der Übertragung eines ausgefüllten Formulars an den Server-Rechner bzw. ein CGI-Programm müssen die Daten so übertragen werden, daß es dem CGI-Script möglich ist zu erkennen, aus welchen Feldern das Formular besteht, und welche Daten der Anwender in welches Feld eingetragen hat. Deshalb gibt es eine bestimmte Kodierungsmethode, die Formularfelder und deren Daten voneinander trennt. Diese Kodierungsmethode benutzt folgende Regeln:

Die einzelnen Formularelemente inclusive ihrer Daten werden durch ein kaufmännisches & voneinander getrennt. Name und Daten eines Formularelements werden durch ein Istgleichzeichen = voneinander getrennt. Leerzeichen in den eingegebenen Daten (z.B. bei mehreren Wörtern) werden durch ein Pluszeichen + ersetzt. Alle Zeichen mit den ASCII-Werten 128 bis 255 (hexadezimal 80 bis FF) werden durch eine Hexadezimalzeichenfolge umschrieben, eingeleitet durch ein Prozentzeichen % und dahinter der Hexadezimalwert des Zeichens (z.B. wird der deutsche Umlaut ö durch %F6 umschrieben). Alle Zeichen, die in diesen Regeln als Steuerzeichen vorkommen (also &, +, = und %) werden ebenfalls hexadezimal umschrieben, und zwar genau so wie höherwertige ASCII-Zeichen.

```
  AnwenderName=Stefan+M%FCnz&AnwenderMail=s.muenz@euromail.com
              &Text=Das+ist+ein+kleiner+Text
```

Erläuterung:

So kodiert der WWW-Browser die Formulardaten beim Absenden des Formulars. Diese Zeichenkette wird mit einer der erlaubten Methoden POST oder GET an das aufgerufene CGI-Script übergeben. Das CGI-Script kann diesen Datenstrom in Kenntnis der Kodierungsregeln auseinanderdividieren, um die Formulardaten beispielsweise feldweise zu verarbeiten.

```
#!/usr/bin/perl
  read(STDIN, $Daten, $ENV{'CONTENT_LENGTH'});
  print "Content-type: text/html\n\n";

  print "<html><head><title>CGI-Feedback</title></head><body>\n";

  @Formularfelder = split(/&/, $Daten);
  foreach  (@Formularfelder)
   {
    ($name, $value) = split(/=/, $_);
    $value =~ tr/+/ /;
    $value =~ s/\%([a-fA-F0-9][a-fA-F0-9])/pack(\"C\", hex(\$1))/eg;
```

```
    print "$name = $value", "<br>\n";
   }

  print "</body></html>\n";
```

A.6.5 Java

Sprachmerkmale

Java wurde vollständig neu entworfen. Die Designer versuchten, die Syntax der Sprachen C und C++ soweit wie möglich nachzuahmen, verzichteten aber auf einen Großteil der komplexen und fehlerträchtigen Merkmale beider Sprachen.

Java ist sowohl eine objektorientierte Programmiersprache in der Tradition von Smalltalk als auch eine klassische imperative Programmiersprache nach dem Vorbild von C. Im Detail unterscheidet sich Java aber recht deutlich von C++, das denselben Anspruch erhebt. Durch die Integration einer großen Anzahl anspruchsvoller Features wie Multithreading, strukturiertem Exceptionhandling oder eingebauten grafischen Fähigkeiten implementiert Java eine Reihe interessanter Neuerungen auf dem Gebiet der Programmiersprachen.

In Java gibt es die meisten elementaren Datentypen, die auch C besitzt. Arrays und Strings sind als Objekte implementiert und sowohl im Compiler als auch im Laufzeitsystem verankert. Methodenlose Strukturtypen wie struct oder union gibt es in Java nicht. Alle primitiven Datentypen sind vorzeichenbehaftet und in ihrer Größe exakt spezifiziert. Java besitzt einen eingebauten logischen Datentyp boolean.

Java bietet semidynamische Arrays, deren initiale Größe zur Laufzeit festgelegt werden kann. Arrays werden als Objekte angesehen, die eine eingeschränkte Menge an Methoden bieten. Mehrdimensionale Arrays werden wie in C dadurch realisiert, daß einfache Arrays ineinander geschachtelt werden. Dabei können auch nicht-rechteckige Arrays erzeugt werden. Alle Array-Zugriffe werden zur Laufzeit auf Einhaltung der Bereichsgrenzen geprüft.

Die Ausdrücke in Java entsprechen weitgehend denen von C und C++. Java besitzt eine if-Anweisung, eine while-, do- und for-Schleife und ein switch-Statement. Es gibt die von C bekannten break- und continue-Anweisungen in normaler und mit einem Label versehenen Form. Letztere ermöglicht es, mehr als eine Schleifengrenze zu überspringen. Java besitzt allerdings kein allgemeines goto-Statement. Variablendeklarationen werden wie in C++ als Anweisungen angesehen und können an beliebiger Stelle innerhalb des Code-Parts eines Programms auftauchen.

Als OOP-Sprache besitzt Java alle Eigenschaften moderner objektorientierter Sprachen. Wie C++ erlaubt Java die Definition von Klassen, aus denen Objekte erzeugt werden können. Objekte werden dabei als Referenzdatentypen behandelt, die wie Variablen angelegt und verwendet werden können. Zur Initialisierung gibt es Konstruktoren, und es kann eine optionale Finalizer-Methode definiert werden, die aufgerufen wird, wenn das Objekt zerstört wird. Seit 1.1 gibt es lokale Klassen, die innerhalb einer anderen Klasse definiert werden.

Applets

Eine der am meisten gebrauchten Erklärungen für den überraschenden Erfolg von Java ist die enge Verbindung der Sprache zum Internet und zum World Wide Web. Mit Hilfe von Java ist es möglich, Programme zu entwickeln, die über das Web verbreitet und mit Hilfe eines Browsers wie Netscape Navigator, Sun HotJava oder Microsoft Internet Explorer gestartet werden können. Dazu wurde die Sprache HTML um das APPLET-Tag erweitert. Sie bietet so die Möglichkeit, kompilierten Java-Code in normale Web-Seiten einzubinden.

Ein Java-fähiger Browser enthält einen Java-Interpreter (die virtuelle Java-Maschine, auch kurz VM genannt) und die Laufzeitbibliothek, die benötigt wird, um die Ausführung des Programms zu unterstützen. Die genaue Beschreibung der virtuellen Maschine ist Bestandteil der Java-Spezifikation, und Java-VMs sind bereits auf eine große Anzahl unterschiedlicher Plattformen portiert worden. Ein Applet kann damit als eine neue Art von Binärprogramm angesehen werden, das über verschiedene Hardware- und Betriebssystemplattformen hinweg portabel ist und auf einfache Weise im Internet verteilt werden kann.

Im Gegensatz zu den eingeschränkten Möglichkeiten, die Script-Sprachen wie JavaScript bieten, sind Applets vollständige Java-Programme, die alle Merkmale der Sprache nutzen können. Insbesondere besitzt ein Applet alle Eigenschaften eines grafischen Ausgabefensters und kann zur Anzeige von Text, Grafik und Dialogelementen verwendet werden. Einer der großen Vorteile von Applets gegenüber herkömmlichen Programmen ist ihre einfache Distributierbarkeit. Anstelle explizit auszuführender Installationsroutinen lädt der ClassLoader des Browsers die Bestandteile eines Applets einfach aus dem Netz und führt sie direkt aus. Das ist vor allem bei kleineren und mittelgroßen Anwendungen in einer lokalen Netzwerkumgebung sehr hilfreich, insbesondere wenn diese sich häufig ändern.

Sicherheit war eines der wichtigsten Designziele bei der Entwicklung von Java, und es gibt eine ganze Reihe von Sicherheitsmechanismen, die verhindern sollen, daß Java-Applets während ihrer Ausführung Schaden anrichten. So ist es einem Applet, das in einem Web-Browser läuft, beispielsweise nicht erlaubt, Dateioperationen auf dem lokalen Rechner durchzuführen oder externe Programme zu starten.

Die wichtigsten Unterschiede kann man in einer kurzen Liste zusammenfassen:

- Das Hauptprogramm eines Applets wird immer aus der Klasse Applet abgeleitet. Bei einer Applikation ist es prinzipiell gleichgültig, woraus die Hauptklasse abgeleitet wird. Eine Applikation wird gestartet, indem vom Java-Interpreter die Klassenmethode main aufgerufen wird. Das Starten eines Applets wird dadurch erreicht, daß der Web-Browser die Applet-Klasse instanziert und die Methoden init und start aufruft.
- Aus Sicherheitsgründen darf ein Applet in der Regel weder auf Dateien des lokalen Rechners zugreifen noch externe Programme auf diesem starten. Eine Ausnahme bilden signierte Applets. Für eine Applikation gelten diese Beschränkungen nicht.
- Ein Applet arbeitet immer grafik- und ereignisorientiert. Bei einer Applikation dagegen ist es möglich, auf die Verwendung des AWT zu verzichten und alle Ein- und Ausgaben textorientiert zu erledigen. Applets bieten einige zusätzliche Möglichkeiten im Bereich des Benutzerschnittstellen-Designs, die so bei Applikationen nicht ohne weiteres zu finden sind. Die Ausgabe von Sound beispielsweise ist standardmäßig auf Applets beschränkt.

```
import java.awt.*;
import java.applet.*;

public class Hello extends Applet {
   void paint(Graphics g) \
       showStatus("Hello, world");
       g.drawString("Hello, world",10,50);
   }
}
```

Das Einbinden eines Applets in ein HTML-Dokument erfolgt unter Verwendung des APPLET-Tags, es wird also durch <APPLET> eingeleitet und durch </APPLET> beendet. Zwischen den beiden Marken kann ein Text stehen, der angezeigt wird, wenn das Applet nicht aufgerufen werden kann. Ein applet-fähiger Browser ignoriert den Text.

Beispiel:

```
<APPLET CODE="Hello.class" WIDTH=300 HEIGHT=200>
 Hier steht das Applet Hello
</APPLET>
```

Servlet

Javasoft hat mit den Servlets - in Anlehnung an Applets auf Clientseite - ein Konzept der portablen (betriebssystem- bzw. prozessorcodeunabhängig) Java-Bytecode-Objekte für Server entwickelt, das die Vorzüge von Applets und CGI auf der Grundlage eines objektorientierten Ansatzes vereint. Das Konzept beschreibt eine einfache, robuste, plattformunabhängige Schnittstelle auf Serverseite, die eine Erweiterung der Funktionalität unabhängig vom Server und von der Plattform mit Java-Bytecode erlaubt.

Servlets sind zu einem spezifischen Interface konforme Java-Objekte. Sie ähneln Applets insofern, als sie - dynamisch über das Netz zu ladende - Bytecode-Objekte sind. Andererseits haben Servlets kein „Gesicht“, das heißt, sie haben kein eigenes grafisches Interface.

Die Vorteile der Servlets gegenüber anderen Server-seitigen Erweiterungen sind:

- Servlets sind schneller als CGI-Skripten
- Sie benutzen eine Standard-Schnittstelle (Servlet-API)
- Servlets sind auf alle Server mit Java-Unterstützung übertragbar, die das Servlets-API unterstützen.
- Sie nutzen alle Vorteile von Java: gute Portierbarkeit, moderne, C++-ähnliche Programmiersprache, gut etablierter Standard.

Servlets sind Protokoll- und Plattformunabhängige, serverseitige Komponenten. Sie sind in Java geschrieben und

erweitern dynamisch alle Server, auf denen Java-Programme laufen. Sie stellen einen Rahmen für das Request-Response-Prinzip zur Verfügung. Dessen ursprünglicher Nutzen ist es, einen sicheren Zugang zu Daten aus HTML-Webseiten zu ermöglichen.

Weil es sich bei den Servlets um serverseitige Programme handelt, benötigen sie keine graphische Benutzerschnittstelle. Servlets stellen das Server-seitige Gegenstück zu Applets dar.

Generell unterstützen Servlets individuelle Protokolle. Sie sind auch flexibel genug, um standardisierte Protokolle, wie HTTP oder HTTPS einzuschließen.

Verwendung von Servlets Servlets können einfach über eine HTML-Seite aufgerufen werden. Die Daten werden über Methoden des HTTP-Servers übermittelt. Weil über Servlets mehrere Requests gleichzeitig bearbeitet werden können, besteht die Möglichkeit der Synchronisation der Anfragen, z.B. für Online-Conferencing. Es kann eine Zusammenarbeit verschiedener Servlets definiert werden, die eine Aufgabenteilung ermöglicht. Servlets können miteinander kommunizieren, auch wenn sie sich auf verschiedenen Servern (und auch Plattformen) befinden.

Wichtigste Methoden

- Init: Beim Aktivieren eines Servlets wird als erstes die Init-Methode ausgeführt. Sie wird im Servlet-Livecycle nur einmal aufgerufen (also nicht bei jedem einzelnen Request). Es empfielt sich, zeitaufwendige Initialisierungsaufgaben in der Init-Methode zu implementieren.

- Service: Mit dieser Methode werden alle Client-Requests behandelt. Die Methodenaufrufe können gleichzeitig erfolgen.
- Destroy: Alle Requests werden solange bearbeitet, bis das Servlet vom Web-Server explizit beendet wurde.

Aufbau von Servlet-Programmen Servlets unterstützen das bekannte Request/Response-Modell. Normalerweise wird das Servlet-Interface durch Erweitern der generischen oder einer HTTP-spezifischen Implementierung erstellt. Das einfachste mögliche Servlet definiert als einzige die Methode Service.

```
import javax.servlet.*;

public class MyServlet extends GenericServlet
{
        public void service (
                ServletRequest  request,
                ServletResponse response
        ) throws ServletException, IOException
        {
                ....
        }
}
```

Die Methode Service stellt die Parameter Request und Response zur Verfügung. Diese enthalten die empfangenen Daten vom Client (Request) bzw. die zu sendenden Antwortdaten für den Client (Response). Die Request- und Responsedaten werden über Input- und Outputstreams bearbeitet.

```
ServletInputStream in = request.getinputStream ();
ServletOutputStream out = response.getOutputStream ();
```

Diese Input- und Outputstreams können für jedes beliebige Datenformat benutzt werden, z.B. zum Austausch von Applet/Servlet-Daten mittels Objektserialisation. Auch HTML und beliebige Bildformate sind geeignet.

Vor- und Nachteile des Servlet-Konzeptes **Vorteile**

- Plattformunabhängigkeit: Servlets können auf jeder beliebigen Plattform ohne erneute Compilierung (Übersetzung) oder Implementierung übertragen werden; (Perl-basierte CGI-Skripten sind von Plattform zu Plattform übertragbar, aber CGI-Servererweiterungen, geschrieben in Hochsprachen wie C sind nicht portierbar.)
- Performance: Servlets müssen nur einmal geladen werden, während CGI-Skripten bei jedem Request neu geladen werden (Ausnahme: Fast-CGI). Die Servlet-Init-Methode erlaubt es, zeit- und resourcenintensive Aktionen bereits bei der Initialisierung auszuführen, d.h. nur einmal im Livecycle des Servlets. Diese Effizienzsteigerung kompensiert die Tatsache, daß Javacode langsamer ist als compilierter Code, bei weitem. Im Vergleich zu Prozessorcode-abhängigen APIs (z.B. NSAPI) sind Servlets signifikant langsamer.
- Java ist robust und völlig objektorientiert. Spezialisierte Javabibliotheken, Entwicklungswerkzeuge und Datenbankmanagementsysteme werden immer gebräuchlicher und sind auch in Servlets verwendbar.

- Eine Programmiersprache für Client- und Serverseitigen Anwendungen. Darstellung der Requestbearbeitung im Vergleich mit CGI und Fast-CGI-Servern.

Nachteile

- Die Servlet-API ist erst im Frühstadium ihrer Entwicklung.
- Javacode ist langsam.

A.6.6 Quellen

HTTP: http://kandinsky.wi.hs-wismar.de/ zimmerma/diplom/node22.html

SELFHTML Stefan Münz Version 7.0 vom 27.04.1998:
http://tkug.telekabel.at/wien/teleweb/selfhtml/selfhtml.htm

Servlets 1998 Schrattenecker Andreas, Stadler Werner, Weilhartner Stefan:
http://infosoft.soft.uni-linz.ac.at/Info/RL/Serverex/servlets.htm

Go to Java 2 Guido Krüger 1999: http://www.gkrueger.com/books/k99a.html

XML Küno Dnhülter 1998: http://members.aol.com/xmldoku

A.7 Klassisches Planen

von Nils Malzahn

A.7.1 Was ist Planen?

Definitionen

Aufgabe, Situation, Aktion, Situationskalkül Die **Aufgabe** des Planens besteht darin, eine Aktionsfolge zu finden, die eine gegebene exakt definierte Ausgangssituation in eine exakt definierte Zielsituation überführt.

Eine **Situation** S ist der Schnappschuss des interessierenden Ausschnitts der Welt zu einem Zeitpunkt, beschrieben durch logische Formeln.

Eine **Aktion** a überführt eine gegebene Situation S in eine andere Situation: die Nachfolgesituation S'. Aktionen können in Operatoren und Tasks unterteilt werden. **Operatoren** sind schematische Beschreibungen von Aktionen. **Tasks** sind Instanzen dieser Operatoren.

Ein **Situationskalkül** ist eine formale Vorschrift, die aus einer gegebenen Situation S und einer Aktion a, die in S ausgeführt wird, die Nachfolgesituation S' berechnet.

formale Definition eines Plans Gegeben sei eine Theorie T, eine Zielformel $Z(s)$ mit einer einzelnen ungebundenen Variable s, dann ist das Planungsziel des klassischen Planens ein Folge von Aktionen $\vec{a}$ zu finden, so daß

$$T \models \text{Legal}(\vec{a}, S_0) \wedge Z(do(\vec{a}, S_0))$$

wobei $do([a_1, \ldots, a_n], s)$ eine Abkürzung für

$$do(a_n, do(a_{n-1}, \ldots, do(a_1, s)) \ldots))$$

ist und $Legal([a_1, \ldots, a_n], s)$ für

$$M\ddot{o}gl(a_1, s) \wedge \ldots \wedge M\ddot{o}gl(a_n, do([a_1, \ldots, a_{n-1}], s))$$

steht. ([Klingspor, 1998])

Teilprobleme des Planens

Das Problem des Planens läßt sich in folgende (tw. überlappende) Teilprobleme aufspalten ([Görz, 1995]):

- Das *qualification problem* ist das Problem, korrekte Vorhersagen über die Fakten nach Ausführung einer Menge von Aktionen zu machen, ohne alle (vorhandene) Information über die Situation vor ihrer Ausführung berücksichtigen zu müssen. Welche Fakten gibt es?
- Das *prediction problem* ist das Problem, vorherzusagen, welche Fakten nach der Ausführung einer Menge von Aktionen wahr sein werden. Welche Fakten werden wahr?
- Das *persistence problem* ist das Problem, vorherzusagen, welche Fakten nach der Ausführung einer Menge von Aktionen unverändert bleiben. Welche Fakten bleiben?
- Das *frame problem* ist das Problem, vorherzusagen, welche Fakten nach der Ausführung einer einzigen Aktion unverändert bleiben.
- Das *ramification problem* ist das Problem, vorherzusagen, welche weiteren Fakten wahr werden, wenn durch die Ausführung einer Aktion in einer Situation ein Faktum wahr wird. Welche Fakten kommen hinzu?

Ein effizienter Planer wird alle fünf Probleme effizient lösen oder umgehen müssen. Da die o.g. Probleme i.A. unentscheidbar sind, muss man sie vereinfachen, um in zumutbarer Zeit Ergebnisse zu erhalten. Kurz: man muss sich mit i.A. nicht vollständigen und nicht korrekten Ergebnissen (Heuristiken) zufrieden geben.

A.7.2 Klassisches Planen

Das klassische Planen schränkt das *qualification problem* bzw. das *frame problem* durch zwei grundlegende Annahmen ein.

1. Aktionen verursachen nur partielle Änderungen der Situation.
2. Aktionen bestehen aus
   (a) den notwendigen Voraussetzungen zur Anwendung der Aktion.
   (b) den Folgen der Aktion.

Je nachdem, ob die Zielsuche im Zustandsraum der möglichen Situationen oder im Raum von partiell bearbeiteten Plänen stattfindet, wird zwischen mengenbasisertem Planen und planbasiertem Planen unterschieden. Eine dritte Form des Planens ist das deduktive Planen, bei dem ein (maschineller) Beweis für die Lösbarkeit des Problems gesucht wird, der dann in einen Plan transformiert werden kann.

mengenbasiertes Planen

Die Situation wird als eine Menge von Prädikaten repräsentiert. Aktionen löschen Prädikate oder fügen welche hinzu. Aktionen sind also Operationen auf Mengen für die effiziente Algorithmen bekannt sind. Operatoren werden (z.B. in STRIPS) als Tripel (V,D,A) notiert.

(V) Die Vorbedingung V beschreibt unter welchen Umständen der Operator angewendet werden kann.

(D) Die Liste D („delete list“) enthält alle Prädikate, die aus dem Weltmodell gelöscht werden müssen, wenn der Operator zur Anwendung kommt.

(A) Die Liste A („add list“) enthält alle Prädikate, die dem Weltmodell hinzugefügt werden müssen, wenn der Operator zur Anwendung kommt.

Durch das Suchen und Probieren von Operatoren, deren Vorbedingungen erfüllt sind, bis das Planungsziel erreicht ist, entwickelt sich ein Plan (die Folge von Tasks, die zum Ziel führte).

deduktives Planen

Grundlage deduktiven Planens sind logische Sprachen, in denen die Planungsdomäne und die Aktionen formal beschrieben (axiomatisiert) werden. Pläne werden ebenfalls durch logische Formaeln spezifiziert. Der Planungsprozess besteht darin einen konstruktiven Beweis für einen gegebene Planspezifikation zu finden. Für prädikatenlogische Verfahren wird häufig das Resolutionskalkül oder die Konnektionsmethode verwendet. Da der Kalkül es erlaubt aus einer Aktion die Effekte abzuleiten, macht das *ramification problem* keine Schwierigkeiten. Das *frame problem* bleibt jedoch ein schwerwiegendes Problem.

planbasiertes Planen

Das Problem des Planens ist der i.A. zu grosse Suchraum. Wenn es möglich wäre ihn sinnvoll zu beschränken, indem z.B. Teile, die mit ziemlicher Sicherheit keinen sinnvollen Plan enthalten einfach abschneidet. Die Bestimmung dieser Teile ist jedoch schwierig. Aus diesem Grund führte Sacerdoti 1975 die Idee ein, auf partiell ausgearbeiteten Plänen zu suchen, die im Laufe der Planungsprozedur verfeinert werden. Es werden für Teilaufgaben Teilpläne entwickelt, die dann ihrerseits wieder in einen übergeordneten Plan eingefügt werden. Pläne deren Vorbedingungen nicht erfüllt werden können, können ausgeschlossen werden, so dass Teile des Suchraums weggeschnitten werden. Der Gesamtplan wird so aus den Teilplänen zusammengefügt.

A.7.3 Beispiele

STanford Research Institute Problem Solver (STRIPS)

STRIPS ist ein mengenbasierter Planer, der den Grundstein für viele weitere Arbeiten auf dem Gebiet der Planung gelegt hat.

In STRIPS ist das Weltmodell durch eine Anzahl von prädikatenlogischen Formeln 1. Ordnung repräsentiert. Operatoren bilden die Grundlage der Lösungen der Probleme. Sollen Roboterpläne entwickelt werden, so gehört zu jedem Operator genau eine vom Roboter ausgeführte Aktion. Um einige Probleme zu umgehen (z.B. das frame problem) trennt STRIPS die Aufgabe des Theorembeweisens von der Suche im mögliche Weltenraum. Die Methoden des maschinellen Beweisens werden nur *innerhalb eines* Weltmodells benutzt, um Informationen darüber zu erhalten, welche Operatoren anwendbar sind bzw. welche Bedingungen im Modell erfüllt sind. Für die Suche im Weltenraum benutzt STRIPS eine „means-end analysis strategy".

Der Suchraum für STRIPS wird durch drei Punkte beschrieben:

1. Ein Start-Weltmodell (Menge von log. Formeln)
2. Eine Menge von Operatoren, ihren Vorbedingungen und Folgen (vgl. I A.7.2)
3. Eine Zielsituation (als log. Formel repräsentiert)

Die Suche erfolgt, indem die Differenz[1] zwischen dem Startmodell S und Zielmodell Z und die Tasks, die diesen Unterschied verkleinern können herausgearbeitet werden. Wenn ein Task T gefunden wurde, nach dessen Anwendung ein oder mehrere Vorbedingungen zur Erreichung des Ziels erfüllt sind, teilt man das ursprüngliche Problem in zwei Teile. Erstens: wie wird das Ziel nach Anwendung von T in S erreicht? Zweitens: welche Taskfolge, von S ausgehend, führt dazu, dass die Vorbedingungen von T erfüllt werden? Diese Teilprobleme werden dann dem Planungsalgorithmus übergeben. Das Zusammensetzen der Teillösungen erzeugt den korrekten Gesamtplan.

`Plan(`S_0`,`Z_0`)`

1. `Bestimme Differenz(`S_0`,`Z_0`).`
   `Keine Differenz`$\rightarrow$`Erfolg, Abbruch.`
2. `Bestimme` T_0 `so, dass Differenz(`S_0`,`Z_0`) kleiner wird`
   `Wenn es kein` T_0 `gibt`$\rightarrow$`Misserfolg, Abbruch.`
3. `Setze` $S_{11} := S_0$, Z_{11} `:=Vorbedingungen von` T_0
4. `Setze` $S_{12} := S_0 \cup T_0$`,` $Z_{12} := Z_0$
5. `Plan(`S_{11}`,`Z_{11}`)`
6. `Plan(`S_{12}`,`Z_{12}`)`

[1] Bedingungen, die in S nicht erfüllt sind, in Z aber erfüllt sein müssen

TWEAK

Das System TWEAK ([Görz, 1995]) ist ein planbasierter Planer. Dieser Planer berechnet den Lösungsplan, indem er vorhandene Pläne durch zusätzliche Tasks erweitert oder einen schon vorhandenen Plan verfeinert. Verfeinern bedeutet, dass der Planer einen Teilplan nochmals unterteilt und so evtl. Umordnungen innerhalb der Tasks vornehmen kann, so dass positive Nebeneffekte erzielt werden. Verfeinern kann auch bedeuten, dass ein bisher noch nicht gelöstes Problem, das sind in diesem Fall noch nicht erfüllte Vorbedingungen für einen Teilplan, in einfacher zu beherrschende Teil-Teilpläne zerteilt werden. Ein Teilplan kann u.U. aus nur einer Task bestehen.

Arbeitsablauf TWEAKs:

1. Initialisierung des Systems mit einem „leeren Plan". Der Plan besteht aus einer Start-Aktion, die als Nachbedingung die Startsituation beinhaltet und einer Ziel-Aktion, deren Vorbedingung das spezifizierte Ziel ist. Die Start-Aktion liegt *vor* der Zielsituation.
2. Verfeinerung bzw. Erweiterung des Plans, solange es noch unerfüllte Operatorvorbedingungen gibt.
   (a) Auswahl eines noch unerfüllten Teilziels.
   (b) Berechnung der Möglichkeiten zur Erreichung dieses Teilziels. Die Berechnung kann folgende Ergebnisse liefern:
      i. Eine schon benutzte Task sichert das Teilziel zu und kann *vor*gezogen werden..
      ii. Es gibt einen Operator, der vor dem Ziel in den Plan eingebracht werden kann.
      iii. Es ist nicht möglich das Teilziel zu erfüllen.
   (c) Auswahl einer möglichen Aktion zur Erreichung des unerfüllten Ziels.
   (d) Einfuegen der entsprechenden Task in den Plan.

Insgesamt erinnert das planbasierte Planen doch sehr der Konfliktserialisierbarkeit von Transaktionen aus der Datenbankwelt.

Planning by Rewriting

Der hier vorgestellte Planer ist ein anytime-Planer, d.h. nach einer kurzen Anlaufphase kann der Algorithmus zu jeder Zeit angehalten und ein korrekter Plan ausgegeben werden. Der grobe Ablauf sieht so aus:

- Wähle einen Initialplan aus (Wie wird dieser Plan effizient erzeugt?)
- Generiere die Nachbarschaft dieses Plans, das ist eine Menge von Pläne die sich durch die Überarbeitungsregeln („rewriting rules") erstellen lassen.
- Bestimme die Kosten mit Hilfe der gegebenen Kostenfunktion.
- Wähle den Plan aus, der als nächstes betrachtet werden soll.

Ein Plan wird als Graph dargestellt. Die Knoten symbolisieren die Aktionen und die Kanten legen eine partielle zeitl. Ordnung (welche Aktion vor welcher anderen durchgeführt werden muss) zwischen den einzelnen Aktionen fest.

Die Schwierigkeiten bestehen darin (schnell) einen Startplan und eine sinnvolle Kostenfunktion zu bestimmen. Bei der Auswahl des nächsten zu betrachtenden Plans scheinen greedy-Strategien gut zu funktionieren. Mit einer max. Suchtiefe pro Schritt kann der größte Erfolg pro Schritt erzielt werden. Das verbraucht jedoch relativ viel Zeit. Demgegenüber steht die Suche nach der „ersten Verbesserung“, die im allgemeinen zwar mehr Überarbeitungsschritte zum „optimalen“ Plan benötigt, dafür erfolgt jede einzelne Verbesserung schneller, so dass evtl. ein besserer Plan in kürzerer Zeit vorgelegt werden kann (anytime-Algo).

Die Überarbeitung erfolgt i.W. durch folgende Maßnahmen:

Umordnen von Aktionen, die nicht voneinander abhängig sind.

Zusammenfassen von mehreren Teilaktionen zu einer Aktion, die alle Teilziele der Teilaktionen erreicht. Statt mehrere Fernabfragen durchzuführen, kann hier z.B. Zeit gespart werden, indem nur eine Abfrage durchgeführt wird.

Aufspalten von Aktionen. Eine Aktion, die mehrere Teilziele erreicht wird in mehrere kleine Schritte aufgespalten. Das ermöglicht es evtl. zeitl. Abhängigkeiten zu entschärfen.

Parallelisieren von Aktionen, die in keiner zeitlichen Abhängigkeit zueinander stehen und deren Ressourcenanforderungen eine parallele Ausführung erlauben.

A.7.4 Grenzen des klassischen Planens

Klassisches Planen

- ist in der Komplexität kaum handhabbar.
- kann nicht mit unvollständigem Wissen (über die Umgebung) umgehen.
- scheitert an sich ändernden Welten.
- kann den Einfluss von Zeit nicht direkt berücksichtigen.

Komplexität

Klassisches Planen in definiter Hornlogik unter Verwendung von Operatoren ohne Vorbedingung und einer Nachbedingung ist bereits PSpace-Vollständig.

unvollständiges Wissen

Mit einem klassischen Planer ist es nicht möglich Fakten zu berücksichtigen, die erst zur Laufzeit bekannt werden. Dem Planer müssen alle Informationen über die Umwelt, das Problem und mögliche Problemlösungen (Aktionen) *vor* Ausführung des Plans bekannt sein. Aus diesem Grund scheitert ein klassischer Planer an sich ändernden Welten, da im Falle einer Änderung kein vollständiges Wissen über die geänderte Welt vorhanden ist. Auch die Möglichkeit Änderungen durch Angabe verschiedener Welten mit unterschiedlichen Warscheinlichkeiten zu berücksichtigen, kann nicht genutzt werden, da der klassische Planer nicht nur vollständiges, sondern auch sicheres Wissen benötigt. Wahrscheinlichkeiten lassen sich nicht in Hornformeln ausdrücken.

Zeit

Zeit kann im klassischen Planen nur durch eine Folge von Vorher- Nachhersituationen bzw. durch eine Reihenfolge von Aktionen dargestellt werden. Eine Änderung der Situation durch das Verstreichen von Zeiteinheiten (etwa 5min) kann nicht berücksichtigt werden ($\rightarrow$ die Welt ändert sich ohne eine entsprechende Aktion).

Lösungsversuche

Um die Beschränkungen, denen das klassische Planen unterliegt, zu brechen, sind verschiedene Ansätze versucht worden. Um Unsicherheiten zu bewältigen wird i.A. eine Erweiterung des Repräsentationsformalismus oder des Situationskalküls (z.B. läßt man mehrere mögliche Welten zu) vorgenommen. Beide Ansätze steigern jedoch die Komplexität des Problems so, dass sie noch weniger handhabbar wird.

Mit „bedingtem“ Planen wird versucht das Problem des unvollständigen Wissens zu lösen. Bei diesem Verfahren treten an die Stelle von unbekannten, erst zur Laufzeit bestimmbaren Umweltbedingungen, Verzweigungsmöglichkeiten und Wahrnehmungshandlungen. Zur Laufzeit wird aufgrund einer Sensorik ein Zweig ausgewählt. Ein zweiter ähnlicher Ansatz ist der Ansatz des **reaktiven Planens**.($\rightarrow$ Ausarbeitung O. Geppert)

A.8 Nichtklassisches Planen

von Oliver Geppert

A.8.1 Was ist nichtklassisches Planen?

Mit *nichtklassischem Planen* bezeichnen wir Planungsansätze, die die Möglichkeiten des *klassischen Planens* erweitern, bzw. dessen Beschränkungen aufzuheben versuchen.

Eines der Hauptprobleme von klassischem Planen ist die Annahme, daß das Wissen des Planers über die Welt vollständig ist und daß Änderungen in der Welt einzig und allein durch den Planer hervorgerufen werden können, deren Erfolg aber nicht überprüft werden kann. Betrachten wir dazu das Beispiel eines Roboters, das auf *Oren Etzioni* zurückgeht:

Man setze einen Roboter vor eine Tür nicht bekannter Farbe. Der Roboter hat Eimer mit blauer und roter Farbe und kann damit Dinge anstreichen. Der Roboter weiß auch, daß er hinter einer roten Tür in die Tiefe stürzen würde. Aufgabe des Roboters ist es, hinter der blauen Tür neu aufzutanken.

Problemstellungen mit klassischem Planen

Da die Farbe der Tür dem Roboter nicht bekannt ist, hat der Planer nur zwei Möglichkeiten, über die Farbe der Tür Sicherheit zu erlangen: durch Anmalen der Tür in Rot oder Blau. War die Tür vorher rot, und er malt sie blau an, dann stürzt er frohen Mutes in den Abgrund, um aufzutanken. War sie blau, und er malt sie rot an, so wird er nie durch die Tür gehen und irgendwann mangels Energie stehenbleiben.

Ein weiteres Beispiel für die Beschränktheit klassischen Planes wäre ein Autopilot, der die Höhe als Summe der seit dem Start ausgeführten Flugmaneuver berechnet hat, wo sich Ungenauigkeiten bei der Ausführung aber zu einer Differenz von mehreren Metern aufaddiert haben: während der Planer noch glaubt, er befinde sich 50 m über dem Erdboden, ist das Flugzeug schon längst unsanft aufgeschlagen.

Einfache nichtklassische Lösungen

Die Lösung für Probleme, die im Fehlerfall bei abstürzenden Robotern und Flugzeugen teure Katastrophen als Ergebnis haben können, lassen sich ziemlich einfach mit Sensoren erreichen: der Roboter kann sehr einfach entscheiden, ob er durch die Tür geht, wenn er vorher untersucht hat, welche Farbe die Tür hat. Genauso kann das Flugzeug sicher landen, wenn es zur Fehlerkorrektur aktuelle Höhenmessungen vornimmt.

Die Möglichkeiten, die Entscheidungen aufgrund von Sensorinformationen bieten, warum natürlich auch den Leuten bekannt, die klassisches Planen gemacht haben. Der Grund, warum klassische Planungssysteme so beschränkt sind, ist ziemlich einfach: den Plan für eine klassische Situation zu finden ist schon schwer genug!

Als Beispiele für nichtklassische Planungssysteme, sollen *reaktives* und *bedingtes* Planen kurz vorgestellt werden.

A.8.2 Reaktives Planen

Reaktives Planen zeichnet sich dadurch aus, daß es auf Sensorinformationen *reagiert*. Im Unterschied zum klassischen Planen hat der Planer a priori keine Informationen über den Zustand, in dem sich seine Umwelt befindet. Allerdings hat er Informationen darüber, was sich in seiner Umwelt befindet bzw. befinden kann. Weiterhin hat der Planer genaue Instruktionen dafür, was er als Reaktion auf eine bestimmte Information auszuführen muß.

Beispiel: die Braitenberg–Fahrzeuge

Eine extreme Variante des reaktiven Planens sind die *B*raitenberg–Fahrzeuge, die auf *Valentin Braitenberg*[Braitenberg, 1984] zurückgehen:

Man stelle sich ein Fahrzeug vor, welches vorne zwei (z. B. lichtempfindliche) Sensoren hat. Angetrieben wird das Fahrzeug über die Hinterräder (als Effektoren). Sensoren und Effektoren sind miteinander gekoppelt und zwar so, daß entweder Sensoren und Effektoren der gleichen Seite oder überkreuz miteinander verbunden sind. Die Sensorinformationen können auf die Effektoren hemmend oder verstärkend wirken (vgl. Abb. I A.1).

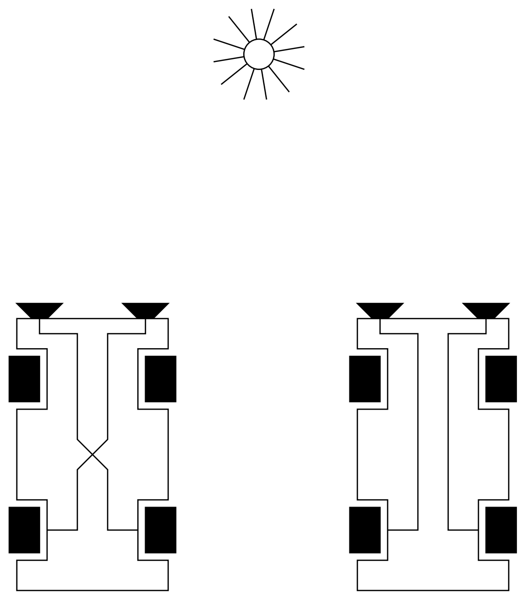


Abbildung A.1: Braitenberg–Fahrzeuge

Es sind vier Szenarien denkbar:

- **Schaltung: überkreuz, Wirkung: hemmend**
  Der der Lichtquelle zugewandte Sensor verlangsamt das gegenüberliegende Rad. Dadurch dreht sich das Fahrzeug von der Lichtquelle weg. Eine Kollision mit dem Hindernis wird vermieden.

- **Schaltung: gleichseitig, Wirkung: hemmend**
  Der der Lichtquelle zugewandte Sensor verlangsamt das gleichseitige Rad. Das Fahrzeug wendet sich der Lichtquelle zu, verringert aber im weiteren die Geschwindikeit, bis es vor dem Hindernis stehen bleibt (um es zu untersuchen vielleicht).

- **Schaltung: überkreuz, Wirkung: verstärkend**
  Der der Lichtquelle zugewandte Sensor beschleunigt das gegenüberliegenbde Rad. Das Fahrzeug wendet sich dem Hindernis zu und beschleunigt. Dem Hindernis gegenüber wird ein recht aggressives Verhalten gezeigt (sogenanntes Fußballspielsyndrom).

- **Schaltung: gleichseitig, Wirkung: verstärkend**
  Der der Lichtquelle zugewandte Sensor beschleunigt das gleichseitige Rad. Dadurch entfernt es sich vom Hindernis je schneller, je näher es an ihm dran ist. Das Fahrzeug flüchtet quasi aus dem Einflußbereich.

Schon dieses einfache Beispiel zeigt recht beeindruckend, wie mit den gleichen Mitteln völlig unterschiedliche Verhaltensmuster zu erreichen sind.

Vor– und Nachteile reaktiven Planens

Hauptvorteil des reaktiven Planens ist seine Geschwindigkeit und Robustheit. Da die Sensoren (mehr oder minder) direkt mit den Effektoren gekoppelt sind, findet keine Planungs– sondern nur noch Entscheidungstätigkeit statt. Pläne müssen nicht erst mühselig aufgestellt und bewiesen werden, da die Reiz–Reaktions–Schemas gespeichert vorgegeben, bzw. fest verdrahtet sind. Ein Planer, der die Aufgabe hat, Kollisionen zu verhindern, kann in beliebige Situationen gesetzt werden, ohne in die neue Umgebung eingewiesen werden zu müssen.

Nachteil des reaktiven Planens ist die Starrheit der Reiz–Reaktions–Schemas. Diese müssen vom Menschen vorgegeben werden und sind nur für einfache Ziele überhaupt machbar. Nicht gegen Hindernisse zu laufen ist einfach im Vergleich zu der Aufgabe, ein Bild aufzuhängen, wenn man nur auf Reize und Reaktionen angewiesen ist.

Einsatzgebiete reaktiven Planens

Einsatzgebiet für reaktive Planer sind hauptsächlich Roboter, Fahrzeuge, Maschinen usw., da sie in der realen Welt zur realen Katastrophen führen können. So sollte eine Maschine, die Gewinde schneidet, nicht erst die Vor– und Nachteile abwägen, ein Gewinde in die vor ihr stehende Person zu schneiden sondern stoppen. Auch sollte ein Roboter, egal was er ausführen wollte, nicht eine Treppe hinunterstürzen.

Ähnlich den Reflexen bei Lebewesen dienen reaktive Planer dazu, Betriebssicherheit zu schaffen und Katastrophen (gleich welcher Art oder Ausprägung) zu verhindern.

Ein Anwendungsfall bei Softbots könnte sein, bei Erhalt eines Terminierungsbefehls nicht noch eine weitere Internetseite anzufordern, sondern unmittelbar nicht gespeicherte Daten zu sichern.

A.8.3 Bedingtes Planen

Bedingtes Planen erweitert das klassische Planen um die Möglichkeit, das Abfragen von Sensorinformationen in Pläne einzubauen und aufgrunddessen verschiedene Wege zu Planen. Man erzeugt also einen Baum von möglichen Plänen, wohingegen man beim klassischen Planen auf einen linearen Plan beschränkt ist.

Die Welt ist für den Planer geschlossen. Er weiß, welchen Objekten er begegnen kann und wie er Informationen über sie abfragen kann. Er weiß auch, welche möglichen Zustände ein Objekt einnehmen kann. Zur Erstellung des Plans ist es nicht notwendig, den aktuellen Zustand der Welt zu kennen. Letztere darf sich auch während der Ausführungsphase ändern, ohne daß die Änderung aus einer Handlung des Planers resultiert.

Wie das klassische Planen auch, ist das bedingte Planen präaktiv. Das heißt, daß eine Aktion erst in allen Möglichkeiten zuende geplant und dann erst ausgeführt wird.

Beispiele: Flughafen und Omelette

Dieses Beispiel geht zurück auf *Hector Levesque* [Levesque, 1996]:

Gegeben ist ein Agent, der sich zu Hause befindet. Aufgabe ist es, zum Flughafen zu gehen und an Bord von Flug 123 zu gehen. Der Flughafen hat zwei Gates, A und B, wobei vorher nicht bekannt ist, an welchen Gate sich Flug 123 befinden wird. Im Flughafen steht eine Abflugtafel, die der Agent lesen kann (vgl. Abb. I A.2.)

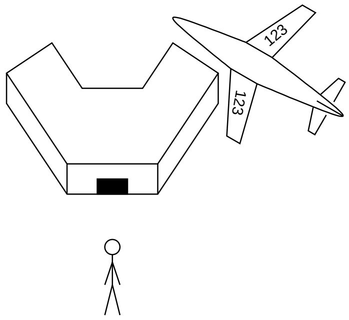


Abbildung A.2: Flughafen

Der Plan zur Erfüllung der Aufgabe läßt sich einfach formalisieren:

geh zum Flughafen;
überprüfe die Abflugtafel;
  if Flug 123 steht an Gate A
    then geh nach Gate A
    else geh nach Gate B;
geh in das Flugzeug;

Etwas komplizierter ist das Omelette–Beispiel, das auch von *Levesque* stammt:

Gegeben sei eine Schüssel, eine Tasse sowie viele Eier, unter denen sich mindestens drei gute befinden. Der Agent hat nun die Aufgabe, dafür zu sorgen, daß sich drei gute Eier in einem der Behältnisse befinden. Dabei kann er durch Riechen erkennen, ob sich in einem Behältnis ein faules Ei befindet. Außerdem kann er Eier in ein beliebiges Behältnis schlagen und den Inhalt der Tasse in die Schüssel transferieren. (vgl. Abb. I A.3.)

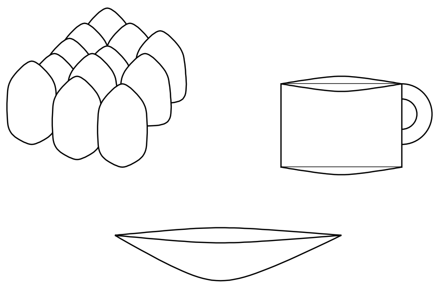


Abbildung A.3: Omelette

Auch hier läßt sich schnell ein geeigneter Plan aufstellen:

until es sind drei Eier in der Schüssel **do**
  until es ist ein Ei in der Tasse **do**

```
        schlage ein Ei in die Tasse;
        rieche an der Tasse;
        if in der Tasse ist ein faules Ei
            then entsorge den Inhalt der Tasse
    end until;
    transferiere den Inhalt der Tasse in die Schüssel;
end until;
```

Vor– und Nachteile bedingten Planens

Klarer Vorteil ist die Flexibilität bedingten Planens. Für jede mögliche (innerhalb der Welt) Gegebenheit bzw. Situation wird der beste Plan ausgeführt, ohne daß der Planbaum neu berechnet werden muß. Weiterer Vorteil ist, daß nicht a priori festehen muß, in welchen Zustand sich die Welt befindet. Zur Qualitätssicherung kann festgestellt werden, ob eine ausgeführt Handlung auch Erfolg hatte.

Nachteil ist, daß das Problem, einen Plan zu finden, durch die Erweiterung klassischen Planens natürlich nicht einfacher geworden ist. Auf unvorhergesehene Ereignisse, die von Außen in die Welt eingreifen, (Mensch tritt dem Roboter in den Weg), kann nicht reagiert werden.

Weiter stellt sich die Frage, wann ein Plan als korrekt anzusehen ist. Beim klassischen Planen ist der Plan genau dann korrekt, wenn er das gewünschte Ziel als Ergebnis hat. Beim bedingten Planen gibt es unterschiedliche Auffassungen, ob alle Wege im Planbaum zum Erfolg führen müssen, oder ob es reicht, wenn nur einige Wege das Ziel erreichen.

Einsatzgebiete bedingten Planens

Bedingtes Planen ist natürlich überall dort einsetzbar, wo Fallunterscheidungen zu treffen sind. Insbesondere ist bedingtes Planen auch für Softbots anwendbar: die Welt, in der sich der Softbot bewegen soll (Computernetze, speziell das Internet), ist ihrer Natur nach beschränkt. Die möglichen Zustände, die ein zu untersuchendes Objekt einnehmen kann, lassen sich immer auf Bits und Bytes reduzieren und sind damit immer interpretierbar.

A.8.4 UWL

Die *University of Washington Language* (UWL) ist eine auf STRIPS basierende Sprache, die hauptsächlich von *Oren Etzioni*[Etzioni et al., 1992] entwickelt wurde. UWL ist eine Sprache, in der bedingte Pläne formuliert werden können. Dazu wurden die Konstrukte, die STRIPS bietet, erweitert, so daß man den Status von Objekten abfragen kann und dadurch bedingt verzweigen.

Wie bei STRIPS gliedern sich die Operationen in UWL in Vor– und Nachbedingungen. Die Vorbedingungen legen fest, welche Gegebenheiten erfüllt sein müssen, damit der entsprechende Operator angewandt werden darf. Dies entspricht direkt den Vorbedingungen von STRIPS. Die Nachbedingungen geben an, welche Aktionen ausgeführ werden dürfen, wenn die Vorbedingungen erfüllt sind. Dabei können Objekte verändert oder deren Status abgefragt werden. Das Ändern von Objekten entspricht dabei den Add– und Deletelisten von STRIPS.

Mögliche Nachbedingungen:

- **cause**
  Mit cause werden Handlungen in der Welt durchgeführt.
  cause (time !clock . 23.00) würde eine Uhr auf 23.00 Uhr stellen

- **observe**
  Observe dient dazu, den Status von Objekten abzufragen oder zu überprüfen, ob ein Prädikat erfüllt ist.
  observe (time !clock . ?Uhrzeit) würde in der Laufzeitvariablen ?Uhrzeit festhalten, welche Zeit die !clock angibt.

Mögliche Vorbedingungen:

- **satisfy**
  Satisfy stellt sicher, daß ein Prädikat wahr oder ein Objekt einen bestimmten Wert hat, wobei cause und observe als Mittel zugelassen sind.
- **find–out**
  Find–Out schränkt satify in dem Sinne ein, daß nur observe–Anweisungen benutzt worden sein dürfen, um die Bedingung zu erfüllen.
- **hands–off**
  Hands–Off schließlich stellt sicher, daß ein Objekt im Laufe des Planpfades nicht durch cause verändert wurde, liefert aber, im Gegensatz zu satisfy und find–out, keinen Wahrheitswert zurück.

Weiter ist UWL gegen STRIPS erweitert, als es Sprachkonstrukte bietet, um Fallunterscheidungen beschreiben zu können.

- **pcond**
  pcond (Tür_ist_offen P1 P2) führt im weiteren Verlauf Plan P1 aus, wenn die Tür tatsächlich offen ist, ansonsten Plan P2.
- **vcond**
  vcond ((!clock = 23.00) P1 P2) führt Plan P1 aus, wenn !clock den Wert 23.00 Uhr hat, ansonsten P2.

Als dritte Erweiterung arbeitet UWL mit drei Wahrheitswerten:

- **T (true)**
  Eine Bedingung ist genau dann T, wenn sie erfüllt ist.
- **F (false)**
  Eine Bedingung ist genau dann F, wenn sie nicht erfüllt ist.
- **U (unknown)**
  Eine Bedingung ist genau dann U, wenn der Planer nicht weiß, ob sie den Wert T oder F hat.

Planen mit UWL

Da UWL STRIPS erweitert, eignen sich, mit Anpassungen, alle Planungsalgorithmen für UWL, die man auch für das klassische Planen verwendet (vgl. das Referat über klassisches Planen). Da die Erzeugung des Planbaums aber mindestens so schwer ist, wie das Erzeugen eines linearen Plans, sind die klassischen Algorithmen für bedingte Pläne nicht schneller, sondern eher langsamer.

Ein Planer, der mit partiell geordneten Plänen arbeitet, wird z. B. von Etzioni vorgestellt.

Die Annahme, das der Wert eines Objektes immer erst verifiziert werden muß (da die Welt zwar als bekannt aber offen angesehen wird), wird von späteren Planern (Etzioni[Etzioni et al., 1997]) wieder eingeschränkt: Teilwelten werden als geschlossen angesehen, so daß man vollständiges Wissen über sie hat. So gibt z. B. das Unix–Kommando ls -a nicht nur Auskunft darüber, welche Dateien sich in einem Verzeichnis befinden, sondern implizit auch darüber, welche Dateien sich **nicht** in dem Verzeichnis befinden.

Der Algorithmus von Etzioni entscheidet die Frage, ob sich die Datei bla.txt im Verzeichnis blubb befindet, stark vereinfacht nach folgendem Schema:

if „blubb/bla.txt existiert nicht“ $\in$ Datenbank
  then return F
if „blubb/bla.txt existiert“ $\notin$ Datenbank $\wedge$ „ls -a blubb“ wurde schon ausgeführt
  then return F
  else return U

Zur Effizienz sei nur angemerkt, daß ein Planer mit geschlossenem Wissen über Teilwelten von 300 zufällig generierten Problemen innerhalb 1000 Prozessorsekunden 94 %, ein Planer ohne geschlossenes Wissen über Teilwelten nur 8 % gelöst hat. Der erste Planer hat dazu weniger als die Hälfte an Aktionen ausgeführt als der zweite.

A.9 Reinforcement Learning

von Volkher Kaschlun

A.9.1 Einführung

Reinforcement learning ist ein beliebtes Modell eines Agenten, welcher durch trial-and-error in Wechselwirkung mit einer dynamischen Umgebung lernt.

Die Unterschiede zum supervised learning bestehen unter anderem darin, daßes keine input/output–Paare gibt, der Agent sofort nach Wahl einer Aktion den "Lohn" und den Folgezustand erfährt, aber nicht erfährt, welches die beste Aktion für maximal lange Laufzeit wäre. Deshalb besteht die Notwendigkeit aktiv Wissen über mögliche Systemzustände, -Aktionen,-Übergänge und -Belohnungen zu sammeln. Eine weiter wichtige Anforderund an das reinforcement learning ist die online-performance. Die Wahl der naechsten Aktion soll in einem beschränkten Zeitraum stattfinden. Außerdem soll es möglichst keine Aufteilung in Lern- und Performance-Phasen geben.

reinforcement learning Modell

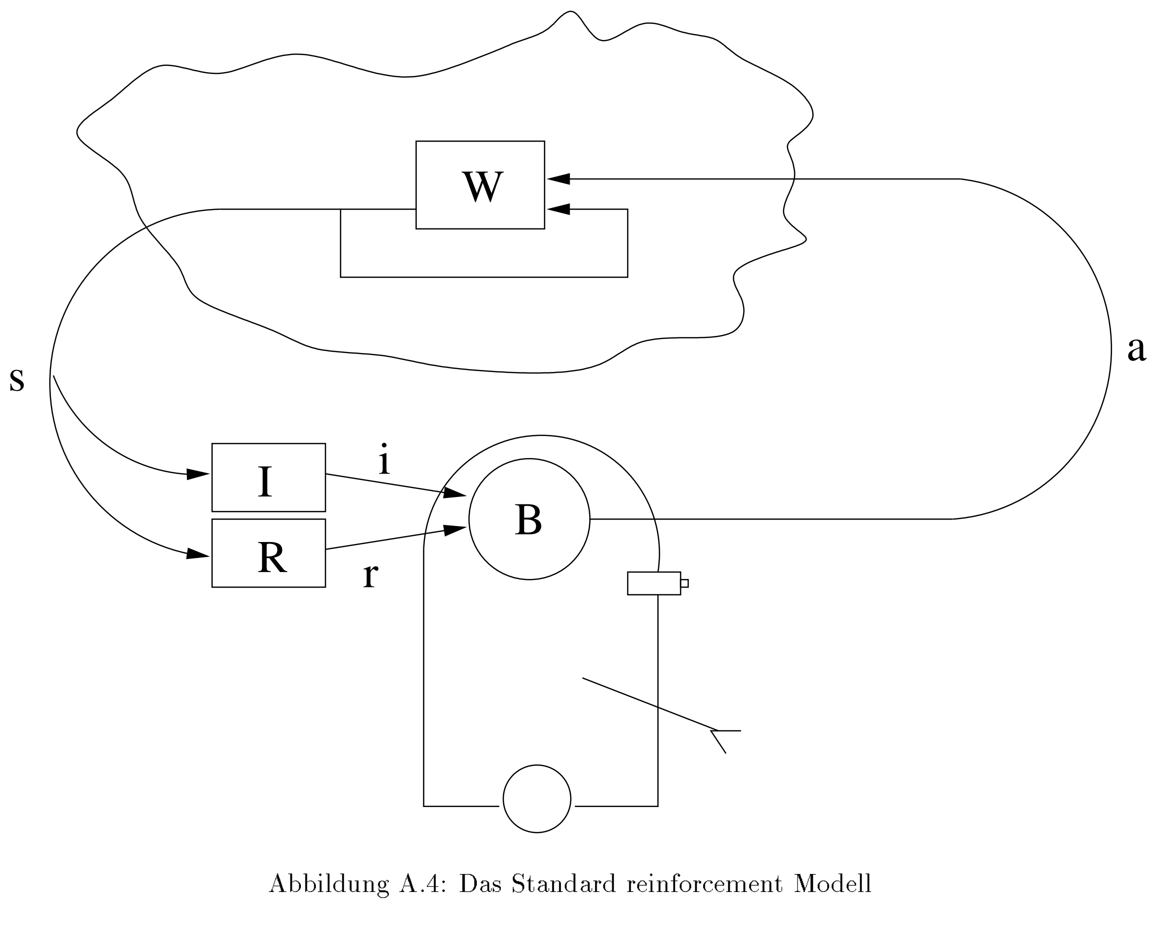


Abbildung A.4: Das Standard reinforcement Modell

Standard-Modell Der Agent ist mit seiner Umgebung via Wahrnehmung und Aktionen verbunden. Er empfängt Hinweise auf den aktuellen Zustand der Umgebung und wählt daraufhin eine als Ausgabe generierte Aktion. Diese Aktion ändert Zustand der Umgebung und der Wert des neuen Zustandes wird dem Agenten als spezieller skalarer *reinforcement input* reflektiert. Ziel des Agenten ist es nun, solche Aktionen zu wählen, die zur Erhöhung der Summe der Werte des reinforcement Signals tendieren. Formal besteht das Model aus

- einer diskreten Menge von Umgebungszuständen S
- einer diskreten Menge von Agenten-Aktionen A
- einer skalaren Menge von reinforcement-Werten, meistens $\{0, 1\}$ oder die reelen Zahlen
- einer Eingabefunktion I, welche die Sicht des Agenten auf den Umgebungszustand bestimmt. Im weiteren nehmen wiran, daß es die Identitätsfunktion ist (d.h. der Agent erkennt den genauen Zustand).

Beispiel	**Umgebung:**	Du bist in Zustand 65. Du hast 4 mögliche Aktionen
	Agent:	Ich nehme Aktion 2.
	Umgebung:	Du erhältst ein reinforcement von 7 Einheiten. Du bist nun im Zustand 15. Du hast 2 mögliche Aktionen
	Agent:	Ich nehme Aktion 1.
	Umgebung:	Du erhältst ein reinforcement von -4 Einheiten. Du bist nun im Zustand 65. Du hast 4 mögliche Aktionen
	Agent:	Ich nehme Aktion 2.
	Umgebung:	Du erhältst ein reinforcement von 5 Einheiten. Du bist nun im Zustand 44. Du hast 5 mögliche Aktionen
	$\vdots$	$\vdots$

Aufgabe des Agenten Die Aufgabe des Agenten ist es eine Taktik $\pi : S \rightarrow A$ zu finden, die einige langlebige Maße des reinforcement maximiert. Wir nehmen im Allgemeinen an, daß die Umgebung nicht deterministisch sein wird, das heißt, gleiche Aktionen im gleichen Zuständen führen zu unterschiedlichen Zuständen und/oder reinforcement-Werten. Dies geschieht auch im obigen Beispiel: Vom Zustand 65 aus produzierte die gewählte Aktion 2 unterschiedliche reinforcement-Werte und unterschiedliche Zustände. Jedoch nehmen wir an, daßdie Umgebung ist fest. das heisßt die Wahrscheinlichkeiten Zustandsveränderungen zu machen oder reinforcement-Werte zu empfangen ändert sich nicht.

A.9.2 Modelle des optimalen Verhaltens

Bevor wir mit Überlegungen über Algorithmen zum Lernen des optimalen Verhaltens beginnen können, müssen wir uns entscheiden, was unser Modell des optimalen Verhaltens sein wird. Es sollen hier 3 Modelle vorgestellt werden.

finite-horizon Modell

Zu einem gegebenen Zeitpunkt soll der Agent seine Belohnungen für die nächsten k-Schritte optimieren:

$$E(\sum_{t=0}^{k} r_t) \tag{A.12}$$

Dieses Modell ist auf zwei Arten nutzbar. In der ersten wird der Agent eine nicht-statische Taktik benutzen, d.h. sie ändert sich mit der Zeit

1. Schritt: wähle k-Schritt optimale Aktion
2. Schritt: wähle k-1-Schritt optimale Aktion

⋮

letzter Schritt: wähle 1-Schritt optimale Aktion und ende

In der zweiten Art handelt der Agent immer nach derselben Taktik und wählt immer die k-Schritt optimale Aktion, aber seine Sichtweite ist durch den Wert von k begrenzt. Das *finite-horizon* Modell kann mehr Rechenzeit benötigen als andere Modelle.Es ist aber gewöhnlich nicht genau bekannt, wieviel Zeit der Agent zur Verfügung hat.

average-case Modell

Der Agent wählt Aktionen, die die durchschnittliche Belohnung optimieren.

$$lim_{k\to\infty} E(\frac{1}{k}\sum_{t=0}^{k} r_t) \tag{A.13}$$

Das dabei entstehende Problem ist, daß es keine Möglichkeit gibt zwischen zwei Strategien zu unterscheiden. Erstere setzt verstärkt auf größere Belohnungen in der Anfangsphase, während die andere ersteres nicht tut. Die Belohnung in der Anfangsphase wird duch die durchschnittliche Leistung überlagert.

infinite-horizon discounted Modell

Dieses Modell verallgemeinert die beiden vorhrigen Modelle Die Belohnung über eine lange Laufzeit des Agenten wird in Betracht gezogen, aber zukünftige Belohnungen werden exponentiell nachlassend entsprechen dem discount-Faktor γ $[0, 1]$ berücksichtigt.

$$E(\sum_{t=1}^{\infty} \gamma^t r_t) \tag{A.14}$$

A.9.3 Bemessen der Lernleistung

Ein Maß für das Optimum ist ein Kriterium, um die Qualität eines Lernalgorithmus zu beurteilen. Es exitieren unterschiedliche Maße.

- letztendliche Konvergenz zum Optimalen:
  Viele Algorithmen haben eine beweisbare Garantie für eine asymptotische Konvergenz zum Optimum. Dies ist aber nicht-vollständig für die Untersuchung der online-performance, weil zum Beispiel ein schnell und preiswert lernender Agent, der nur 99 % des Optimums erreicht, oft besser ist als ein langsam Lernender, der garantiert 100 % erreicht.
- Konvergenzgeschwindigkeit zum Optimum:
  Das Optimum ist gewöhnlich ein asymptotische Ergebniss; deshalb ist die Konvergenzgeschwindigkeit ein schlechtes Maß.
- Regret ist das Maß der zu erwartenden Abnahme der Belohnung aufgrund der Ausführung des Lernalgorithmus, anstatt sich von Anfang an optimal zu verhalten. Es bezieht sich auf die Idee der *mistake bounds* und bestraft Fehler, wo immer sie auch während der Laufzeit auftreten.

A.9.4 Exploitation versus Exploration

Verwendung des Gelernten versus Lernen während der Verwendung

Beispiel: the two-armed bandit

Ein Agent befindet sich in einem Raum mit zwei Spielautomaten (einarmiger Bandit). Es ist kein Guthaben zum Spielen nötig. Wenn der Hebel i betätigt, zahlt der Automat i 1 oder 0, bezüglich eines Wahrscheinlichkeitsparameter p_i, aus. Welche Strategie soll der Agent nun verfolgen, d.h. welcher Hebel in Schritt t, bei einer gegebenen Liste der zuvor gewählten Hebel und ihrer Auszahlungen, soll ausgewählt werden?

Der Agent kann glauben, daß ein Hebel ziemlich hohe Auszahlungswahrscheinlichkeit hat. Soll er diesen die ganze Zeit wählen oder einen anderen, über welchen wenig Information existiert, der aber schlecht zu sein scheint? Je länger das Spiel dauert, um so schlechter sind die Konsequenzen eines frühzeitigen Konvergierenz zu einem suboptimalen Hebel! Deshalb sollte der Agent um so mehr lernen!

Es existiert eine große Spanbreite von Lösungen, wie *Bayerian Approach*, *Minimax*, *Greedy-Strategien*, *ad hoc randomized Strategien* und vielen anderen.

A.9.5 Delayed reinforcement

Markov Decision Processes

Die Probleme mit dem delayed reinforcement sind gut modelliert als *Markov decision processes*(MDP). Ein MDP besteht aus:

- einer endlichen Menge von Zuständen S
- einer endlichen Menge von Aktionen A
- einer Belohnungsfunktion $R : S \times A \rightarrow \Re$
- einer Zustandsüberführungsfunktion
  $T : S \times A \rightarrow \Pi(S)$, mit $\Pi(S)$ ist eine Menge von Wahrscheinlichkeitsverteilungen über die Menge S.

Finden einer Taktik bei gegebenem Modell

Definition Der optimale Wert eine Zustandes ist die erwartete unendlich abnehmende Summe von Belohnungen, welch der Agent erhält, wenn er in diesem Zustand startet und die optimale Taktik ausführt.

$$V^*(s) = max_\pi E(\sum_{t=0}^{\infty} \gamma^t r_t). \tag{A.15}$$

Diese Optimumswertfunktion ist eindeutig und kann rekursiv definiert werden

$$V^*(s) = max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')), \tag{A.16}$$

. Die optimale Taktik ist dann

$$\pi^*(s) = arg\ max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s'). \tag{A.17}$$

Werte Iteration Algorithmus *Werte Iteration*:

initialisiere $V(S)$ willkürlich
loop
  loop für $s \in S$
    $V(s) := max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$
  end loop
end loop

Eine approximale Optimumsfunktion bringt eine approximal gute Taktik hervor:

$$|V^* - \hat{V}|_{sup} \leq \epsilon \Rightarrow |V^* - V_{\hat{\pi}}|_{sup} \leq 2\epsilon \frac{\gamma}{1-\gamma} \tag{A.18}$$

Es ist aber nicht eindeutig, wann der Werte Iterations Algorithmus beendet werden kann. Es existiert aber das *Bellman residual* Theorem: Wenn die Differenz zweier aufeinanderfolgender Wertefunktionen kleiner als ϵ ist, ist die maximale Differenz einer solchen Wertefunktion und der optimalen Wertefunktion $2\epsilon\gamma/(1-\gamma)$. Die Kosten des Algorithmus sind polynomiell in der Anzahl der Zustände.

Taktik Iteration Algorithmus *Taktik Iteration*:

wähle eine beliebige Taktik π
loop
  berechne den Wert der Taktik-Funktion π:
  $V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s,a,s')V_\pi(s'))$
  verbessere die Taktik in jedem Zustand:
  $\pi'(s) := arg\ max_a$
    $(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_\pi(s'))$
  $\pi := \pi'$
bis keine Verbesserungen mehr möglich sind.

Lernen einer Taktik ohne Anfangsmodell

Die bisherige Methoden betrachteten wir unter der Annahme eines bereits existierenden Modells. Ein solches Modell besteht aus dem Wissen über

1. die Wahrscheinlichkeitsfunktion der Zustandsüberführung $T(s,a,s')$

2. die reinforcement Funktion $R(s,a)$

Ein solches Modell ist aber im vorraus nicht bekannt! Stattdessen soll der Agent mit der realen Welt interagieren, um Informationen zu erhalten, die mit den Möglichkeiten eines geigneten Algorithmus verarbeitet werden können, um eine optimale Taktik zu erzeugen. Es gibt nun zwei Wege um fortzufahren. Beim ersten, auch model-based genanntem Weg, lernt der Agent ein Modell, und benutzt es, um einen Controller abzuleiten. Beim Model-free, lernt der Agent einen Controller, ohne ein Modell zu lernen.

Model-free learning

Das größte Problem, das uns begegnet ist das *temporal credit assignment*. Ist die Aktion, die wir gerade nehmen, eine gute, wenn sie weitreichende Auswirkungen haben kann? In langlebigen Anwendungen ist es schwierig zu wissen, was das "Ende" ist. So kann man nicht bis dahin warten, um die gewählten Aktionen entsprechend ihrem Endergebniss zu belohnen bzw. zu bestrafen. Stattdessen nutzen wir das Verständis von der Wert-Iteration, indem wir den Wert eines Zustandes aufgrund seiner unmittelbaren Belohnung und dem Wert des Folgezustandes (*temporal-difference metods*) verändern.

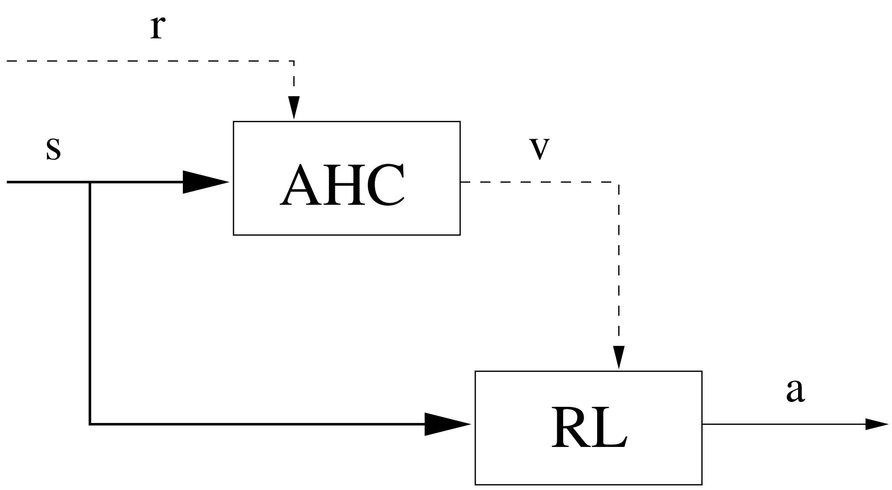


Abbildung A.5: adaptive heuristic critic Architektur

AHC und TD Der *adaptive heuristic critic* Algorithmus ist eine lernende Analogie der Taktik Iteration und besteht aus zwei Komponenten: Einer critic (AHC) und einer reinforcement learning (RL) Komponente. RL kann eine Instanz von jeder der k-armed-Bandit Algorithmen mit einer Modifikation für multiple Zustände sein. Statt die Maximierung der Belohnungen wird die Maximierung der heuristischen Werte v, die von AHC berechnet werden, verwendet. Das AHC benutzt die realen externen reinforcement-Werte, um zu lernen, Situationen mit ihren Werten unter Ausführung der aktuell in RL instanzierten Taktik, aufzuzeichnen. Die Analogie zur Taktik Iteration kann man sehen, wenn wir uns vorstellen, daß die Komponenten alternierend arbeiten[1]. Die Taktik π, durch RL erfüllt, ist fest und AHC lernt die Wertefunktion V_π für diese Taktik. Danach lernt RL mit festem AHC eine neue Taktik π', welche die neue Wertefunktion maximiert.

Suttons TD Algorithmus Es bleibt zu klären, wie das AHC die Werte einer Taktik lernen kann. Es benutzt dazu *Suttons TD Algorithmus* [Sutton, 1988] mit der gewichteten Erneuerungsregel

$$V(s) := (1-\alpha)V(s) + \alpha(r + \gamma V(s')). \tag{A.19}$$

Immer wenn ein Zustand s besucht wird, wird sein Wert erneuert, um dichter an $r + \gamma V(s')$ zu sein, mit

- r ist die instanzierte erhaltene Belohnung,
- $V(s')$ ist der Wert des aktuellen Folgezustandes und

[1] in den meisten Implementierungen arbeiten die Komponenten gleichzeitig

- α ist die Lernrate.

Die motivierende Idee ist, daß $r + \gamma V(s')$ eine Stichprobe für den Wert von $V(s)$ ist. Wenn die Lernrate α richtig eingestellt ist - sie muß langsam abnehmen - garantiert TD ein konvergieren zur optimalen Wertefunktion.

Die TD-Regel ist eine Instanz für eine allgemeine Klasse von Algorithmen, genannt $TD(\lambda)$, mit $\lambda = 0$. TD(0) ist nur in der Lage bei Schätzungen von Wertveränderungen einen Schritt weit zu blicken. Die TD(0)-Regel kann algebraisch umgeschrieben werden zu

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s)); \tag{A.20}$$

die allgemeine $TD(\lambda)$-Regel ist ähnlich,

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s))e(s), \tag{A.21}$$

wird aber eher auf jeden Zustand entsprechend seiner Wählbarkeit e(s), als nur auf den unmittelbar vorherigen Zustand angewendet. Der Wählbarkeitspfad ist dann

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} I(s = s_k), \tag{A.22}$$

mit I ist eine Indikatorfunktion.

Dieser Pfad bezeichnet den Grad der Besuche des Zustands s in der jüngsten Vergangenheit. Bei Empfang eines reinforcment wird es benutzt, um alle Zustände, die kürzlich besucht wurden, entsprechend ihrer Wählbarkeit zu erneuern. $\lambda = 0$ ist gleichbedeutend mit TD(0). $\lambda = 1$ ist grob gesagt gleichwertig zum Belohnen aller Zustände in Bezug auf ihren Wert am Ende der Laufzeit.

$TD(\lambda)$ ist rechnerisch aufwendiger, aber konvergiert erheblich schneller für große λ.

Q-learning Die Arbeit der zwei Komponenten des AHC in einer einheitlichen Weise kann durch Watkins' Q-learning [Watkins, 1989, Watkins and Dayan, 1992] geleistet werden. Um Q-learning zu verstehen, brauchen wir einige zusätzliche Notationen. Sei $Q^*(s,a)$ das erwartete abnehmende reinforcement, mit Aktion a in Situation s, dann fahre fort mit der der Wahl von Aktionen, um Q^* zu maximieren. Dies kann rekursiv geschrieben als

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') max_{a'} \, Q^*(s',a'). \tag{A.23}$$

Watkins hat gezeigt, daß $V^*(s) = max_a \, Q^*(s,a)$ und somit dieses $\pi^*(s) = arg \; max_a \, Q^*(s,a)$ die optimale Taktik ist.

Wir können die Q-Werte on-line mit einer Methode wie bei TD schätzen, die aber auch verwendet wird, um die Taktik zu definieren. Eine Aktion kann gewählt werden, indem man die mit maximalem Q-Wert für die aktuelle Situation nimmt. Die Q-Lernregel lautet

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a'} \, Q(s',a')) \tag{A.24}$$

und wird angewendet, wenn eine Aktin a in einer Situation s mit dem Ergebniss der unmittelbaren Belohnung r und dem nächsten Zustand s' gewählt wird. Wenn jede Aktion in jedem Zustand unendlich oft in unendlicher Laufzeit ausgeführt und α passend verringert wird, werden die Q-Werte mit Wahrscheinlichkeit 1 gegen Q^* konvergieren.

Q-learning kann verallgemeinert werden, um Zustände zu erneuern, die wie in $TD(\lambda)$ früher als einen Schritt vorher auftraten.

A.9.6 Fragen zur praktische Anwendung

Wichtige Fragen, die bei der praktischen Anwendung zu beachten sind:

- Wie wichtig ist die optimale Erkundung?
  Können wir die Lernphase in eine Erkundungs- und eine Anwendungphase aufspalten?
- Was ist die beste Form der langlebigen Belohnungsfunktion:

  Finite horizon?

  Discounted?

  Infinite horizon?

- Wieviel Berechnungspotential steht zwischen den Entscheidungen des Agenten zur Verfügung, und wie soll es genutzt werden?
- Was für früheres Wissen können wir in das System einbauen, und welche Algorithmen sind geeignet, dieses Wissen zu nutzen?

A.10 CiteSeer und Cora

von Niels Schröter

A.10.1 Citeseer[Giles et al., 1998][Bollaker et al., 1998]

Die Suche nach wissenschaftlichen Artikeln kann oft sehr müselig sein, da sie im Web oft schlecht organisiert sind. Oft gibt es die Dokumente nur als Postscript Dateien, die man mit herkömmlichen Suchmaschinen nicht finden kann, da diese nur Text- oder Htmldateien in ihren Index aufnehmen.

Citeseer verfolgt den Ansatz, dem Benutzer die langweilige und umständliche Sucharbeit abzunehmen. Üblicherweise wird zur Suche von Dokumenten Suchmaschinen benutzt, und dann per Hand weitergesucht. Citeseer ist speziell auf die Suche wissenschaftlicher Artikel abgestimmt. Im Unterschied zu Altavista werden allerdings auch Postscriptdokumente berücktichtigt. Die einzige Möglichkeit relevante Postscriptdokumente mit herkömmlichen Suchmaschinen zu finden, liegt darin, nach den Schlüsselwörtern zu suchen, und dann auf den gefundenen links per Hand nach Postscriptdokumenten umzuschauen. Da allerdings wissenschaftliche Publikationen normalerweise in Postscript abgelegt werden, kann dies die Suche stark erschweren.

Citeseer arbeitet wie herkömmliche Suchmaschinen mit einem Webinterface, in das Schlüsselwörter eingegeben werden können. Citeseer gibt die Schlüsselwörter dann an Suchmaschinen wie Altavista oder Hotbot weiter. In gefundenen Dokumenten wird dann nach Begriffen wie "publication" oder "postscript" gesucht. Werden dann Dokumente mit der Endung "ps", "ps.Z" oder "ps.gz" gefunden, werden diese heruntergeladen und in einer Datenbank gespeichtert, die dann wieder über das Webinterface abfragbar ist.

Citeseer benutzt Textähnlichkeitsmessung in 2 verschiedenen Punkten, einmal um Zitatverweise auf Gleichheit zu überprüfen, die verschiendene Sonderzeichen benutzen bzw. ausfühlich sind. Der zweite Punkt Prüfung auf Ähnlichkeit eines gefundenen Dokuments zu anderen thematisch ähnlichen Dokumenten, die Citeseer ebenfalls dem Benutzer als relevante Dokumente anbietet. Für die Ähnlichkeitsprüfung benutzt Citeseer die Methoden "string distance", "LikeIt" und "TFIDF" (term frequency $\times$ inverse document frequency).

String Distance misst die Ähnlichkeit von 2 Dokumenten dadurch, dass gezählt wird, wieviele Einfüge-, Lösch- und Änderungsoperationen Nötig sind, um das eine Dokument in das andere umzuwandeln. Wird dabei ein bestimmter Wert nicht überschritten, werden die Dokumente als gleich klassifiziert. Die Methode String Distance wird jedoch von Citeseer nicht benutzt.

Die Suchmaschine verwendet hingegen LikeIt und TFIDF. TFIDF reduziert bei der Ähnlichkeitsüberprüfung alle Wörter auf ihre Wortstämme; "Computing" und "Computer" werden dadurch zu "Comput". Für das Thema irrelevante Wörter wie "der", "die" werden weggelassen, um den Vergleich weiter zu vereinfachen und um die Genauigkeit zu erhöhen.

Bereits heruntergeladene Dokumente werden zunächst mit Hilfe des Programms pstotext in ASCII Format konvertiert und dann durch einen Parser geschickt, der wichtige Dokumentfeatures extrahiert. Dazu gehören im wesentlichen Header, Abstract, Introduction. Beim Parsing werden bis jetzt allerdings nur englischsprachige Dokumente unterstützt. Das charakteristische Merkmal von Citeseer ist jedoch, das alle Zitatverweise und die komplette Bibliographie mit in die Datenbank aufgenommen werden und dadurch ein Zitatindex erstellt wird. Dieser Zitatindex kann anschliessend mit dem Webinterface durchsucht werden, alternativ zu der direkten Suche in den Dokumenten. Da Zitate von verschiedenen Dokumenten oft leicht im Format variieren, klassifiziert Citeseer mit ICG (Identical Citation Grouping) Zitatverweise auf gleich oder ungleich, um doppelte Einträge in der Zitatdatenbank zu vermeiden.

Die Relevanz eines Dokuments wird daran gemessen, wie oft dieses Dokument von anderen Dokumenten zitiert wird. Sucht der Benutzer ähnliche Dokumente zu einem gefundenen, kann er bidirektional zitierende und zitierte Dokumente zu seinem gefundenen Dokument direkt aufrufen.

A.10.2 Cora[McCallum et al., 1999]

Cora arbeitet im Prinzip ähnlich wie Citeseer, unterscheidet sich jedoch in einigen wichtigen Punkten. Zunächst ist Cora eine Domänenspezifische Suchmaschine, mit der man nur Artikel aus dem Bereich Informatik suchen kann. Ausserdem befragt Cora keine anderen Suchmaschinen, die Suche nach neuen relevanten Dokumenten beginnt immer auf Informatikhomepages von Universitäten. Gleich ist leider jedoch, dass auch von Cora lediglich englischsprachige Dokumente unterstützt werden, was sich aber evtl. in Zukunft ändern könnte.

Die Motivation zu Cora entstand dadurch, dass es relativ umständlich war, mit Suchmaschinen wie Altavista gezielt nach Postscriptdokumenten aus der Informatik zu suchen. Herkömmliche Suchmaschinen verfügen zwar über eine grosse Anzahl an Dokumenten, die Suchpräzision ist allerdings sehr gering.

Eine andere Hürde war immer, das eine präzise Suchmaschine zwar Wünschenswert wäre, der Aufbau einer solchen sich allerdings immer als ziemlich kompliziert herausstellte. Viel Aufwand bedeutet immer auch einen hohen Kostenaufwand.

Ziel bei Cora war es, den Aufwand der Erstellung der Suchmaschine auf ein Minimum zu reduzieren. Cora benutzt dazu Techniken des Maschinellen lernens, um den Aufbau der Datenbank weitgehend zu automatisieren. Dazu gehören "Reinforcement Learning", "Textklassifikation" und "Information Extraction".

Um Dokumente für die Datenbank zu beziehen, benutzen Suchmaschinen Unteragenten, die Spider oder Crawler genannt werden. Cora benutzt eine intelligente Spider, die nicht einfach alle Dokumente bezieht, die sich ihr in den Weg stellen, sondern nur die relevanten.

Die Coraspider wird dazu in einem Reinforcement Learning Umfeld erstellt, damit kann das verhalten der Spider optimal abgestimmt werden. Reinforcement Spider sind ca. 3x effizienter als herkömmliche Spider, die nach Breitensuche vorgehen.

Ähnlich wie bei Yahoo bietet Cora zusätzlich zur Schlüsselwortsuche eine hierarchische Anordnung, die man auf der Suche nach Dokumenten durchstöbern kann. Der Unterschied ist jedoch, dass neu von Cora gefunden Dokumente vollautomatisch in die fest vorgegebene 51-elementige Hierarchie einsortiert werden. Dazu benutzt Cora Textklassifikation.

A.11 Wissensrepräsentation: Frames

von Christian Fischbach

A.11.1 Einleitung

Das Paradigma der objekt-orientierten, operationalen Repräsentation von Weltwissen wurde maßgeblich aufgespannt durch *semantische Netze* und *Frames.*[1] Einsichten in deren Unzulänglichkeiten führten schließlich zur Entwicklung von *KL-ONE*, dem ersten Vertreter semantisch fundierter hybrider Systeme, die auch als Termsubsumtionsformalismen oder terminologische Logiken bezeichnet werden.

Historischer Ursprung

Frames wurden Mitte der achtziger Jahre von Marvin Minsky durch dessen sog. „frames paper" [Minsky, 1974] ins Leben gerufen und erlebten dann eine stürmische Entwicklung. Minskys Motivation bestand insbesondere darin, die Wirksamkeit des gesunden Menschenverstands für Probleme der realen Welt zu erklären. Seine Grundvorstellung war dabei, daß Weltwissen in Packeten, den *frames*, verpackt ist und diese zur Verarbeitung in ein *retrieval-Netz* eingebettet sind. Situationsbedingt wird ein geeignetes Frame aktiviert, das lokal und über seine Verweise zu anderen Frames alles relevante Wissen zur Bewältigung der Situation zur Verfügung stellt. Minskys Vorschläge zur Implementierung waren allerdings weitgehend informal.

Zur Terminologie

Einige KI-Lehrbücher stellen Ideen und Konzepte der Frame-Theorie, teilweise ohne Rückgriff auf die genannten Begriffe, unter anderer Terminologie vor, etwa Schema, Prototyp, Unit und memory organization.

Einige Begriffe sind durch ihren Gebrauch etwas unscharf geworden, so bezeichnet *frame theorie* nicht nur die theoretische Entwicklung aus Minskys frames paper, sondern existiert auch als Synonym für Wissensrepräsentation (WR) auf höherem Level. In ähnlicher Weise wird *frame system* sowohl konkret auf Minskys retrieval-Netz als auch allgemein auf die Implemetierung irgendeiner Frame-Sprache bezogen, die u.U. nur noch recht wenig mit Minskys retrieval-Netz zu tun hat. Gerade auch die mit dem Attribut *frame-basiert* versehenen KL-ONE-Nachfolger haben nur noch wenig Ähnlichkeit mit ursprünglichen Frames.

A.11.2 Higher Level Knowledge Structure

Frame-Theorie, also Wissensrepräsentation auf höherem Level durch Frames, wurde teils parallel teils ohne direkten Bezug zu konkreten Frame-Sprachen entwickelt. Die drei folgenden Abschnitte nennen wichtige sprachunabhängige Motive der Frame-Theorie.

Frame-driven recognition

Die Orientierung am Menschen mit seiner Art der „Problemlösung durch gesunden Menschenverstand" führte zu einer Vorstellung von frame-gesteuerter Erkennung und Bewältigung von Situationen, die stark mit populär psychologischen Begriffen arbeitet. Insbesondere Phänomenfolgen

[1] Diese Zusammenfassung orientiert sich überwiegend an [Maida, 1987], weitere Ergänzungen stammen aus Abschnitt 3.3 in [Morik, 1997] und [Nebel, 1987].

aus den Bereichen visuelle Wahrnehmung (z.B. Betreten eines Raumes) und Textverstehen (z.B. stereotype Ereignissequenzen - sog. scripts - wie Restaurantbesuch, verschiedene Aspekte von Reanalyse) wurden im Frame-Kontext mit Hilfe von Begriffen wie Erkennung, Interpretation, Vorhersage, Überraschung, Desorientierung und Reinterpretation beschrieben.

Frame-Organisation

Die Verknüpfung der Frames untereinander erfolgt wie bereits bei Bartlets Schemata [Bartlet, 1961] nach dem (experimentell beim Menschen verifizierten) Prinzip, daß neue Informationen auf der Basis von vorhandenen Strukturen organisiert und erinnert werden. Die Möglichkeit, situationsspezifisch *passende* Frames zu aktivieren, darf dabei als entscheidender Aspekt der Frame-Organisation angesehen werden. Der Umfang der einzelnen Frames ist nicht vom jeweils repräsentierten Inhalt abhängig, sondern von der Modularisierbarkeit der Komponenten. Wie mehrfach verwendete Komponentenmengen erkannt und zu Modulen abstrahiert werden, ist weitgehend unklar.

ISA-Hierarchien

Frame-Theorie geht davon aus, daß Frames (Konzepte) sich in einer Generalisierungshierarchie befinden. Diese ursprünglich nur implizit enthaltene Annahme wurde in späteren Phasen durch sog. ISA-Hierarchien expliziert, bei denen ein Frame über einen ISA-Slot mit seinen Oberframe(s) verbunden ist. Da ein Frame mehrere Oberframes haben kann, ist die Frame-Hierarchie nicht notwendigerweise baumartig, sondern i.A. eine Halbordnung.

Zum Erwerb dieser ISA-Hierarchie existieren verschiedene Standunkte: Am menschlichen Lernen orientiert ist der Drei-Kategorien Ansatz von Rosch in [Rosch, 1975], der Frames in die zuerst gelernte, wahrnehmungsbasierte Kategorie basic (Bsp. Stuhl) und die durch Spezialisierung bzw. Generalisierung inferierten Kategorien sub- bzw. superordinate (Bsp. Lehnstuhl bzw. Möbel) unterteilt. Innerhalb der Frame-Theorie wurde allerdings der epistemologische Standpunkt bevorzugt, so daß es Aufgabe von Menschen ist, die ISA-Hierarchie top-down zu entwerfen.

A.11.3 Frame-Sprachen

Die in Abschnitt I A.11.2 beschriebenen ISA-Hierarchien stellen das Bindeglied von Frame-Theorie und -Sprachen dar.

Verbreiteter Standard

Obwohl sich die vielen entwickelten Frame-Sprachen im Detail mehr oder weniger stark voneinander unterscheiden, hat sich ein Kern von Konstrukten als verbreiteter Standard herausgebildet.

Diese Kernkonstrukte geben Frames den Charakter von *records*, die obligatorisch einen *ISA-Slot* und frame-spezifische weitere Slots haben. Durch Belegung des ISA-Slots mit dem (Namen des) Oberframe wird ein Frame in die ISA-Hierarchie eingeordnet. Die frame-spezifischen Slots haben (a) einen definierten Wertebereich (sind also *getypt*), (b) optional einen *Default-Wert*, (c) optional *assoziierte Prozeduren*, wobei *if-added*-Prozeduren bei Belegung und *if-needed*-Prozeduren bei Abfrage des Slots ausgeführt werden, und werden (d) entlang der ISA-Hierarchie *vererbt*.

Das Semantikproblem

Ein allgemeines Problem von WR-Formalismen in den achtziger Jahren war ihre fehlende Fundierung mit *formaler Semantik*. Dies betraf insbesondere auch semantische Netze und Frames und äußerte sich in vielen unergiebigen Versuchen, verschiedene Systeme und ihre Eigenschaften miteinander zu vergleichen. Drei wesentlichen Gründe für diese fehlende semantische Fundierung sollen hier genannt werden:

- Minsky selbst hielt Logik aus prinzipiellen Gründen für ungeeignet zur Beschreibung von Frames. Beispielsweise, so Minsky, sei die logische Deduktion auf inhaltlicher Ebene „interessenlos“ und könne deshalb nicht modellieren, wie Frames die Erkenntnis in eine bestimmte Richtung steuern.
- Viel Unklarheit konzentriert sich auf die *semantischen Primtive*: So konnte man sich nicht auf ein festes (minimales) Inventar von Slot- bzw. Kantentypen einigen (viele erweisen sich als sachbereichsabhängig), die Interpretation der Slots (Kanten) war nur möglich durch Intuitionen, die auf dem Wissen über englische Sprache basieren („pretend it's english“, [Hayes, 1985]). Desweiteren blieb undeutlich, ob Frames (Konzepte) intensional, also durch sie definierende Eigenschaften, oder extensional durch die Menge ihrer Instanzen zu interpretieren sind.
- Schließlich wurden in den Diskussionen verschiedene Beschreibungsebenen durcheinadergeworfen, auf denen man über die Systeme sprechen kann: Sprachliche-, begriffliche- logische- und Implementierungsebene. Zum Teil war dieses Durcheinander einigen sprachlichen Konstrukten inhärent, etwa den assoziierten Prozeduren, die Formalismus- und Implementierungsebene vermischten.

Auch Versuche, solche Systeme quasi „im Nachhinein“ durch Angabe von Übersetzungsschemata mit einer formalen Logik zu beschreiben, konnten die Situation nicht wesentlich verbessern, da man zusätzlich zur reinen Prädikatenlogik ein wenig handhabbares Gemisch aus Logikprogrammierung (für die assoziierten Prozeduren), nicht-monotoner Logik (für die Defaults) und getypter Logik (für Type-Checking und ISA-Hierarchie) brauchte.

Hybride Systeme

Gerade auch die in Abschnitt I A.11.3 angesprochenen semantischen Probleme führten Brachman, Levesque und Schmolze[2] zur Entwicklung von KL-ONE, Ausgangspunkt für viele weitere semantisch fundierte hybride Systeme. Grundidee hybrider Systeme ist, Wissen nach Wissensarten getrennt zu repräsentieren, aber integriert zu verarbeiten. Drei wesentliche Forschritte von KL-ONE im Vergleich zu semantischen Netzen und Frames seien im folgenden genannt.

(1) Terminologisches (definitorisches, begriffliches) Wissen, die sog. *T-Box*, wird streng getrennt von Assertionen (Individuen, Instanzen), der sog. *A-Box*. Verarbeitung in der T-Box ist immer Begriffsklassifikation, in der A-Box immer Prüfen (Realisieren) der Instanzeigenschaft.

(2) Statt sachbereichsabhängiger semantischer Primitive werden sachbereichsunabhängige, nur auf das Definieren bezogene *epistemische Primitive* vorgegeben und mit einer *formalen mengentheoretischen Semantik* versehen. Prozeduren zur Verarbeitung müssen dann also nur noch für diese epistemischen Primitive geschrieben werden. Durch die semantische Fundierung werden Systeme und ihre Eigenschaften unabhängig von einer konkreten Implementierung vergleichbar. Den Kern von KL-ONE bildet Termsubsumption für Begriffe und Rollen. Zusätzlich gibt es Wertebereiche von Rollen, Anzahlrestriktionen und Schnittmengenbildung bei Begriffen und Rollen.

(3) Die unter (1) und (2) genannten Punkte ermöglichten formale Komplexitätsabschätzungen für Algorithmen zur Subsumptionsrelation. Treten in den Definitionen keine Zyklen und keine konstruierten Rollen auf, ist das subsumes-Problem entscheidbar in polynomieller Zeit über der Länge der verglichenen Terme. Bereits die Hinzunahme der einfachen Schnittbildung über Rollen macht den Algorithmus entweder unvollständig ($t \gg u \not\Rightarrow \mathit{subsumes}(t,u) = \mathit{1}$) oder nicht mehr

[2] Siehe [Brachman, 1977, Brachman and Schmolze, 1985, Brachman and Levesque, 1985].

polynomiell. Das Problem wird unentscheidbar, wenn beliebige Rollenkonstruktion erlaubt ist. Fazit in der KI ist, die Unvollständigkeit zu akzeptieren oder den Formalismus zu beschränken.

Verhältnis zu OOP-Sprachen

Frame-Sprachen hatten einen großen Einfluß auf die Entwicklung von OOP-Sprachen und sind ihnen sehr ähnlich. Die Unterschiede liegen in der Schwerpunktsetzung: hier die praktische Standardprogrammierung (OOP), dort die Konstruktion von AI-Datenbanken (Frame-Sprachen). So können Frame-Systeme typische objekt-orientierte Simulationen realsieren, wenn auch nicht so effizient.

Anhang B

Ergebnisse der Evaluation

B.1 Auswertung der Ergebnisse

B.1.1 Best-Case-Eingaben

	Name				***Branche***				***Produkt***			***Stichwort***	
Agent	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**gesamt**	**Wert**	**Zeit**
	Antik-Conrady				*Antiquitäten*				*Tische*			*—–*	
MetaGer	2										20	5,000	0.30
Firmen–Agent	4	0	0	0	0	0	0	0	0	4	52	1,300	15.34
	Hochtief				*Bauwesen*				*Software*			*—–*	
MetaGer	2	1	0	1	1	1	2	4	1	0	69	1,725	0.30
Firmen–Agent	1	4	4	0	4	4	0	0	0	2	124	3,100	17.25
	Eiffel				*Computer*				*Compiler*			*—–*	
MetaGer	0	1	2	2	2	1	2	2	2	2	76	1,900	0.30
Firmen–Agent	4	0	1	0	4						72	3,600	18.29
	Maxdata				*Computer*				*Notebook*			*—–*	
MetaGer	1	0	4	1	1	1	4	1	1	1	82	2,050	0.30
Firmen–Agent	4	4	0	0							76	1,900	1.48
	Epson				*Drucken*				*Tintenstrahl-drucker*			*—–*	
MetaGer	4	1	0	0	0	0	0	4	1	0	62	1,550	0.30
Firmen–Agent	4	4									76	3,800	18.20
	—–				*Essen + Trinken*				*Milchshake*			*—–*	
MetaGer	4	1	0	0	0	0	0	0	0	0	49	1,225	0.30
Firmen–Agent	1	1	0	0	1	2	1	0	0	1	40	1,000	11.09
	Nikon				*Fotografie*				*Spiegelreflex-kamera*			*—–*	
MetaGer	4	4	4	0	4	4	0	0	4	4	164	4,100	0.30
Firmen–Agent	1	1	2	4	4						76	2,530	14.35
	Vedes				*Geschenke*				*—–*			*Spielzeugladen*	
MetaGer	2	2	2	2	2	2	0	0	0	2	92	2,300	0.30
Firmen–Agent	1	1	2	4	4						87	2,175	11.23
	IAA				*Konferenzen + Messen*				*—–*			*Auto*	
MetaGer	0	0	1	1	0	0	0	0	0	0	15	0,370	0.30
Firmen–Agent	0	0	0	0	0	4					20	0,330	2.38
	Fendt				*Landwirtschaft*				*Traktor*			*—–*	
MetaGer	0	4	1	0	1	1	0	1	2	1	63	1,575	0.30
Firmen–Agent	0	1	0	1	4	1	1				49	1,750	19.20

	Name				***Branche***				***Produkt***			***Stichwort***	
Agent	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**gesamt**	**Wert**	**Zeit**
	FAZ				*Nachrichten + Medien*				*Tageszeitung*			—–	
MetaGer	2	2	2	1	2	2	2	2	2	2	101	2,525	0.30
Firmen–Agent	4	0	0	0	2	0					52	2,100	8.17
	ZDF				*Nachrichten + Medien*				*heute*			—–	
MetaGer	2	1	0	0	0	2	2	2	4	2	63	1,575	0.30
Firmen–Agent	4	0	2	2	0	0	0				70	2,500	18.39
	Rollerblade				*Outdoor*				*Inlineskates*			—–	
MetaGer	2	2	1	0	0						46	2,300	0.30
Firmen–Agent	4	0	2	2	2	2	2	2	4		116	5,440	34.02
	SPD				*Politik + Verwaltung*				*Steuerreform*			—–	
MetaGer	0	1	1	2	1	1	2	0	1	1	34	2,100	0.30
Firmen–Agent	2										20	5,000	21.03
	Mattel				*Spielzeug*				*Barby*			—–	
MetaGer	1	0									10	1,250	0.30
Firmen–Agent	4	0									40	5,000	25.01
	Telekom				*Telekommu-nikation*				*D1*			—–	
MetaGer	1	0	1	0	0	1	1	0	1	0	29	0,725	0.30
Firmen–Agent	4	0	0	0	1	0	0	0	1	0	18	1.200	20.10
	Alphatelecom				*Telekommu-nikation*				—–			*Telefongesellschaft*	
MetaGer	2	2	0	0							38	2,380	0.30
Firmen–Agent	4	4	1	4	0						112	5,600	17.19
	Greenpeace				*Umwelt*				*Demo*			—–	
MetaGer	4	4	1	4	0	0	1	4	4	1	139	3,475	0.30
Firmen–Agent	0	0	0	0	0	0	0		4	0	8	0,200	8.58
	Tetrapack				*Verpackungen*				*Milchtüte*			—–	
MetaGer	1	0	0	0							10	0,620	0.30
Firmen–Agent	4	2	4								90	7,500	20.10
	Enerko				*Politik + Verwaltung*				—–			*Acropolis*	
MetaGer	2	0	0	0	0	0	0	0	0	0	20	0,500	0.30
Firmen–Agent	4	0									40	5,000	0.33
	Siemens				*Telekommu-nikation*				*IC35*			—–	
MetaGer	1	1	0	1	0	0	1	2			36	1,125	0.30
Firmen–Agent	2	4									56	7,000	19.21

B.1.2 Average-Case-Eingabe

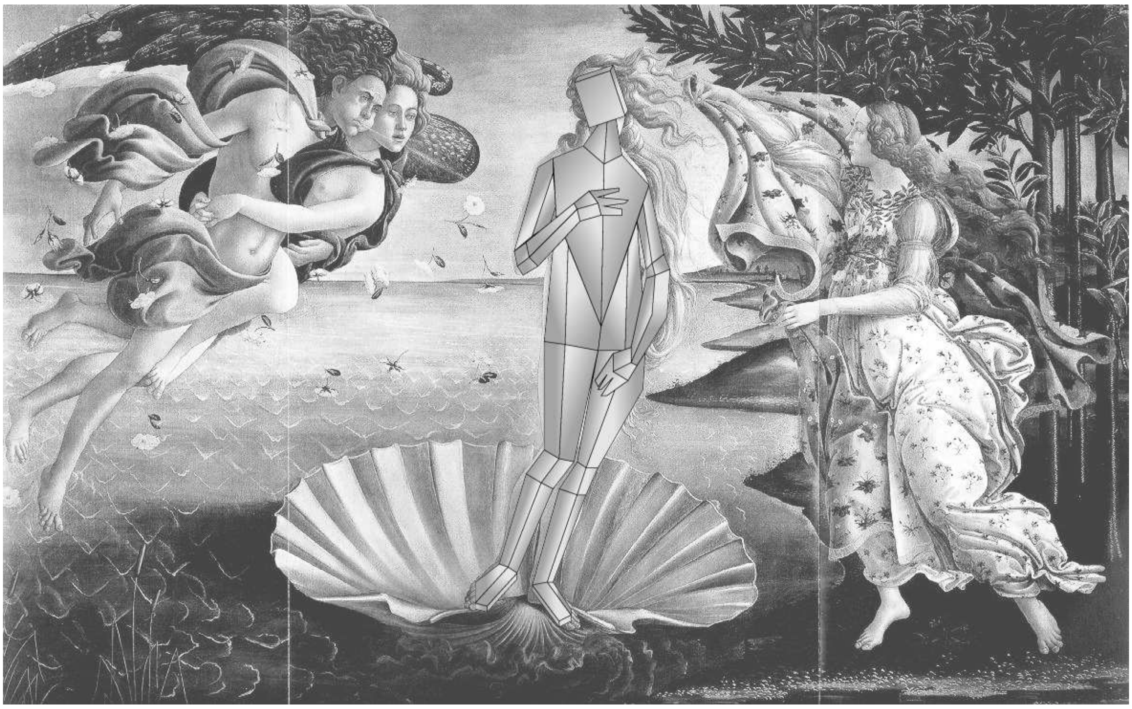
	Name				***Branche***				***Produkt***			***Stichwort***	
Agent	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**gesamt**	**Wert**	**Zeit**
	Beos				*Computer*				*Stinger*			—–	
MetaGer	0	0	0	0	0	0	0	0	0	0	0	0,000	0.30
Firmen–Agent	4	0	2	2	0	0	0	4	0	0	82	2.050	4.33
	Ameling				*Essen + Trinken*				—–			—–	
MetaGer	2	2	0	0	0	0	0	0	0	0	38	0,950	0.30
Firmen–Agent	0	0	2	0	0	1	0	0	0	0	21	0,525	4.24
	—–				*Geschenke*				*Parfumzerstäuber*			—–	
MetaGer	0	0	0	0	0	0	0	0	0	0	0	0,000	0.30
Firmen–Agent	0	0	0	0	0	0	0	0	0	0	0	0,000	4.04
	Niedermeyer				*Fotografie*				—–			—–	
MetaGer	2	2	2	0	0	0	2	0	0	2	70	1,750	0.30
Firmen–Agent	0	0	0	2	0	1					19	0,790	2.05
	Panorama Vision				*Fotografie*				—–			—–	
MetaGer	2	0	4	0	2	0	0	0	0	0	64	1,600	0.30
Firmen–Agent	0	0	0	2	4						38	1,900	3.05
	I-Tüpfelchen				*Geschenke*				—–			—–	
MetaGer	4	0	0	0	0	0	0				40	1,430	0.30
Firmen–Agent	0	0	0	0	0	0	0				0	0,000	3.01
	Classic Link				*Marketing*				*Voice Response System*			—–	
MetaGer	2	0	0	0	0	0	0	0	0	0	20	0,500	0.30
Firmen–Agent	2	0	0	0	0	0	0	2	0		26	0,720	107.35
	Wundermann Cato Johnson				*Marketing*				—–			—–	
MetaGer	0	2	0	0	0	0	0	1	4	0	30	0,750	0.30
Firmen–Agent	1	1	4								51	4,250	2.35
	Skyship				*Reisen*				—–			—–	
MetaGer	2	0	0	2	2	2	2	2	2	2	76	1,900	0.30
Firmen–Agent	0	0	0	4	0	0	0	0	2	2	34	0,850	11.30
	Nicolaus				*Verpackungen*				*Faltschachtel*			—–	
MetaGer	1	1	1	1	1	1	0	0	0	0	46	1,150	0.30
Firmen–Agent	2	0	0	0							20	1,250	13.18
	Thermotemp				*Wissenschaften*				—–			—–	
MetaGer	4	4	4	4	4	4	2	2	0	0	162	4,500	0.30
Firmen–Agent	0	0	4	0	0	4	2	0	0		60	1,660	3.36
	Pieter v. Weenen				*Information*				—–			—–	
MetaGer	2	0									20	2,500	0.30
Firmen–Agent	2	4	4	0							88	5,500	1.05
	All.ex				*Landwirtschaft*				*Taubenabwehr*			—–	
MetaGer	2	0									20	2,500	0.30
Firmen–Agent	4	0	0	0	2	0	0	2	0	4	62	1,550	129.31
	Athener Zeitung				*Nachrichten + Medien*				*Zeitung*			—–	
MetaGer	2	2	2	0	2	2	2	2	2	2	96	2,400	0.30
Firmen–Agent	0	0	2	0	2	1	0	0	4	0	41	1,025	573.26

B.1.3 Worst-Case-Eingaben

Name	Branche	Produkt	Stichwort	gefundene URLs MetaGer Zahl	gefundene URLs MetaGer Zeit	gefundene URLs Firmen–Agent Zahl	gefundene URLs Firmen–Agent Zeit
Antik-Daniel	Antiquitäten	Möbel	—–	3	0.30	2	79.00
Interkrenn	Bauwesen	Bohrmaschine	—–	0	0.30	4	15.20
SAP	Computer	R/4	—–	10	0.30	2	26.45
Trioptimum	Computer	—–	Hardware	0	0.30	0	23.03
Rocketlauncher	Computer	—–	Microsoft	10	0.30	6	30.25
Druckerei Zellerhoff	Drucken	—–	—–	1	0.30	5	7.03
Aldi	Essen + Trinken	Nutoka	—–	1	0.30	2	28.26
Foto-Ullrich	Fotografie	Objektiv	—–	4	0.30	2	10.07
Vatikan	Konferenzen + Messen	—–	Heilige Messe	10	0.30	3	25.00
Essing	Outdoor	Fahrrad	—–	2	0.30	11	30.00
Nokia	Telekommuni-kation	1234	—–	10	0.30	5	30.00

Teil II

Benutzungshandbuch BOTISHELLY



Inhaltsangabe Teil II

Kapitel 1

Einleitung und Überblick

Nicht Kunst und Wissenschaft allein,
Geduld will bei dem Werke sein.
(Mephisto)

Johann Wolfgang Goethe
Faust I, Hexenküche

Bei BotIshelly handelt es sich um eine Java–Klassenbibliothek, mit deren Hilfe Sie eine Hülle (engl. *shell*) für einen Internetagenten, auch *WebBot* genannt, zusammenbauen können. Die Bestandteile dieser Hülle nehmen Ihnen schon sehr viel Verwaltungsarbeit ab, die für die meisten Einsatzgebiete gleich ist, sind aber darauf angewiesen, daß Sie die Shell mit dem für Ihr Gebiet wichtigen Wissen und den notwendigen Werkzeugen versehen. Nur dadurch kann aus der Shell auch ein Agent werden.

Wenn Sie z. B. einen Agenten für die Suche nach bestimmten Softwarepaketen schreiben möchten, so müssen Sie zum einen den Bereich beschreiben und für den Agenten verständlich modellieren, in dem die Suche laufen soll. Das kann die Modellierung der Webseiten von Universitäten sein, oder vielleicht die Seiten von Softwareherstellern. Zum anderen müssen Sie den Agenten mit Werkzeugen (*Operatoren*) ausstatten, um Webseiten analysieren und verarbeiten zu können. So kann ein Operator, der Informationen aus den Seiten einer Universität holt, sich völlig von einem unterscheiden, der auf kommerziellen Seiten arbeitet. Generell gilt, je allgemeiner Sie Ihren Agenten gestalten wollen, desto größer ist der Aufwand, den Sie in die Modellierung des Weltwissens und der Operatoren stecken müssen!

Um eine Suchanfrage zu bearbeiten, verfolgt BotIshelly ein einfaches und intuitives Ablaufschema, das von Ihnen bei Ihren eigenen Recherchen im Internet vielleicht auch schon angewandt worden ist. In der nachstehenden Tabelle werden diese Schritte direkt Klassen von BotIshelly zugeordnet, die selbst (oder davon abgeleitete Klassen) für die entsprechende Bearbeitung zuständig sind:

1.	Entgegennehmen der Benutzeranfrage	`io.InputmaskServlet`
2.	Auswerten und Strukturieren der Benutzeranfrage	`informationexchange.PlanInformation`
3.	Erstellen eines Plans	`planner.Planner`
4.	Ausführen des Plans	`planner.PlanExecution`
5.	Auswerten und Strukturieren der Ergebnisse	`informationexchange.PlanInformation`
6.	Rückgeben der Ergebnisse an den Benutzer	`io.OutputmaskServlet`
7.	(evtl.) Lernen aus der Suche	`learners.Learner`

Das `InputmaskServlet` hat neben der Eingabeanforderung vom Benutzer noch eine zweite, wichtige Aufgabe: das Erzeugen und Starten einer Instanz der Klasse `systemcontrol.FlowControl`. Die `FlowControl` realisiert die zentrale Systemkontrolle für eine Suche und ist u. a. zuständig dafür, daß Planer und Planausführer erzeugt, mit Informationen versehen und angestoßen werden, sowie für die Verwaltung der zur Verfügung stehenden Systemresourcen. Die Hauptklassen von BotIShelly, also Klassen, die direkt von der Systemkontrolle instantiiert werden, wie `Planner` und `PlanExecution`, kommunizieren nie unmittelbar miteinander, sondern immer durch das Eventhandling der Systemkontrolle

Einige wichtige Klassen, ohne die wohl kaum eine Suche Ihres Agenten Erfolg haben wird, sind in obigem Ablaufschema nicht genannt. Diese Klassen beinhalten das oben angesprochene Weltwissen, sowie die Operatoren, um mit der Welt, die Sie beschrieben haben, umgehen zu können. Dabei müssen Sie das Wissen in die Klassen `knowledge.T_Box` und `knowledge.A_Box` nur füllen, wohingegen Sie die Operatoren aus der Klasse `operators.Operator` tatsächlich ableiten und implementieren. Da die Operatoren oft Seiten oder Dateien aus dem Internet holen oder auch klassifizieren müssen, gibt es noch zwei Untergruppen von Klassen, die teilweise Bestandteil von BotIShelly sind, teilweise von Ihnen bearbeitet oder gar neu geschrieben werden müssen, abhängig vom Aufgabengebiet Ihres Agenten. Zur ersten Gruppe gehören die Klassen, die Daten aus dem Internet beschaffen. Zu diesen zählen z. B. die Hierarchie der von `dataprovider.net.NetService` abgeleiteten Klassen, die eine beliebige URL aus dem Netz laden, oder die zu `dataprovider.search.SearchService` gehörenden Ableitungen, die auf die Abfrage von bestimmten Suchmaschinen spezialisiert sind. Zur zweiten Gruppe gehören die Klassen, die Objekte der Welt (d. i. die Suchdomäne), z. B. URLs, klassifizieren können. Diese Klassen leiten sich aus `dataanalysis.classifiers.Classifier` ab.

Dieses Handbuch stellt Ihnen eine Beschreibung der Klassen von BotIShelly zur Verfügung, die über die Javadoc–Dokumentation[1] hinausgeht. Sie sollten die einzelnen Kapitel alle einmal gelesen haben, bevor Sie sich daran begeben, einen eigenen Agenten zu entwickeln. Um Ihnen das Wiederauffinden von wichtigen Textstellen zu erleichtern, benutzen wir drei Arten von Randmarkierungen:

1. **Unbedingt notwendig!**
   In der Textpassage werden Vorgänge beschrieben, die Sie unbedingt machen müssen, wollen Sie einen Agenten schreiben. Dabei kann es sich um Kleinigkeiten, wie das Setzen einer Variable handeln, oder um so komplexe Dinge, wie das Füllen der T–Box.

2. **Achtung!**
   Dies Icon markiert Stellen, an denen Sie etwas beachten müssen. So sind z. B. Hinweise darauf markiert, was passieren kann, wenn Sie an einer bestimmten Variablen drehen.

3. **Ich will mehr!**
   Diese dritte Markierung schließlich zeigt Ihnen, an welchen Stellen von BotIShelly Sie ansetzen können, um tiefergehende als die Standardeingriffe am System vorzunehmen. Dazu zählen z. B. die Erstellung zusammengesetzter Operatoren, oder die Neuentwicklung eines Planers.

Wenn es wichtig ist, zwischen Instanzen und Klassen zu Unterscheiden, benutzen wir eine Kursivschrift für ***`Java-Instanzen`***, eine Schreibmaschinenschrift für `Klassen`. Quellkode und Methodennamen sind auch in Schreibmaschinenschrift gehalten.

Wir haben uns dazu entschieden, Sie in diesem Handbuch entweder direkt oder als *Shellbenutzer* anzureden. Mit „Shellbenutzer" meinen wir eine Person, die die Bilbiothek BotIShelly dazu benutzt, einen Agenten zu schreiben. Im Gegensatz dazu bezeichnen wir Personen, die den fertigen Agenten später benutzen, als *Agentenbenutzer*.

[1] In der Javadoc–Dokumentation finden sie eine ausführliche Beschreibung der einzelnen Klassen und ihrer Schnittstellen.

Kapitel 2

Benötigte Tools und Libraries Dritter

Zu einer Revolution der deutschen Arbeiter kommt es wahrscheinlich erst, wenn Neckermann die aufblasbare Taschenbarrikade anbietet.
Binsenweisheit

2.1 Tools

Java Die Programmierung und die Ausführung des Java-Codes erfolgten mit dem Java 2 SDK, Version 1.2.1. Unter [Sun, 2000] findet sich hier stets die aktuelle Version. Grundsätzliches über Java beantwortet die Site [Javasoft, 2000].

JSDK Das Java Servlet Development Kit wird von uns in der Version 2.0 eingesetzt. Verwendet wird es von der Systemkontrolle/IO, um die Ausgaben zum Browser via Servlets zu realisieren. Weitere Informationen dazu gibt es unter [jwebserver, 2000].

2.2 Libraries

Java Network FTP Library Die Java Network FTP Library wird von der Datenbeschaffung benutzt, um den FTPNetService zu realisieren. Sie unterliegt der GPL und ist zu finden unter [Sheng–Te, 2000]. Eine Versionsnummer ist nicht angegeben.

COM.STEVESOFT.PAT Dieses Pakte wird ebenfalls von der Datenbeschaffung benutzt, das Reguläre Ausdrücke unter Java ermöglicht. Wir haben die Version 1.3.2 eingesetzt; aktuelle Updates scheint es nicht mehr zu geben. Die URL [Stevesoft, 1998] bietet neben verschiedenen Tutorials und Beispielprogrammen auch Zugriff auf das Paket.

2.3 Benötigte Hardware

Für BotIshelly wird folgende Hardware empfohlen:

UltraSparc 10 mit 300Mhz Taktfrequenz

128 MB Hauptspeicher

500MB freier Plattenplatz

Auf anderen Plattformen wurde das korrekte Arbeiten der Shell nicht getestet. Für den Multiuser-Betrieb wird erheblich mehr Hauptspeicher benötigt.

2.4 Installation der Shell

Um BotIshelly zu installieren, führen Sie bitte folgende Schritte durch:

Zunächst wird ein installiertes JDK (Java Developement Kit) ab Version 1.2.0 benötigt (Java 2). Um die vorgegebene Ein-/Ausgabeschnittstelle von BotIshelly zu benutzen, wird außerdem das JSDK2.0 (Java Servlet Developement Kit) benötigt. Beides ist auf `http://java.sun.com` erhältlich.

Nun müssen die Dateien der Shell in ein beliebiges Verzeichnis kopiert werden. Dafür bietet sich z. B. das Homeverzeichnis an. Es entsteht ein Verzeichnisbaum, dessen oberstes Verzeichnis `sourcen` heißt. Der Java-Classpath muß nun zusätzlich auf das Verzeichnis `Shell` zeigen, welches ein Unterverzeichnis von `sourcen` ist, damit der Java-Compiler die entsprechenden Dateien findet. Ist Ihr Homeverzeichnis `/home/foo` und haben sie dort BotIShelly ausgepackt, so muss der Classpath zusätzlich den Pfad `/home/foo/sourcen/Shell` enthalten.

Sie sollten sich nun im Verzeichnis `sourcen` ein Unterverzeichnis anlegen, in dem sich nur die modifizierten Java-Dateien der Shell befinden. Dieses könnte z.B. den Namen `MyShell` tragen. So können Sie sicherstellen, daß die Java-Dateien der Shell unberührt bleiben. Sie müssen dann allerdings sicherstellen das der Java-Classpath nun zuerst auf `MyShell` zeigt und danach erst auf `Shell`. So berücksichtigt der Java-Compiler erst die Java-Dateien aus ihrer angepassten Shell und danach aus BotIshelly. Dies hat den Vorteil, das sie mehrere Agenten parallel entwickeln können, ohne die Übersicht zu verlieren.

Im Verzeichnis Shell befindet sich auch noch ein `Makefile`. Die Entwickler von BotIshelly haben folgende Befehle bereits implementiert: `make all` compiliert alle Java-Dateien. `make run` startet den Servletrunner. `make newrun` tut das selbe wie `make run`, mit dem Unterschied, daß zusätzlich das Logfile gelöscht wird.

Danach kann der Agent getestet werden, in dem in einem Javascipt-fähigem Webbrowser die URL `http://rechnername:8080/servlet/agent` aufgerufen wird. Wichtig ist, daß die Dateien `Shell/config.sys` und `Shell/io/servlet.properties` angepaßt werden, da diese hartcodierte Pfadangaben enthalten.

Kapitel 3

Systemkontrolle und I/O

Was wär ich
ohne dich,
Freund Publikum?
All mein Empfinden Selbstgespräch,
all meine Freude stumm.

Johann Wolfgang Goethe
Der Autor

3.1 Aufgabe und Definition

Da I/O und Systemkontrolle stark integriert sind, werden sie in einem Kapitel beschrieben. Die I/O ist die Schnittstelle zum Agentenbenutzer, d.h. alle Interaktionen zwischen dem Benutzer und dem Agenten werden nur über die `I/O` stattfinden. Die Aufgabe der `I/O` ist es, Eingaben vom Benutzer entgegenzunehmen, sie zur weiteren Verarbeitung an die zuständigen Klassen weiterzuleiten, die Ergebnisse zur Ausgabe vorzubereiten und ausgeben. Außerdem sollte der Benutzer über die Fehler in der Eingabe oder über das Scheitern der Suche in Kenntnis gesetzt werden.

Die Systemkontrolle stellt die Kommunikation zwischen verschiedenen Programmteilen wie z.B. dem Plann[er und der Planausführung sicher, verwaltet die Suchergebnisse, verteilt die Plan- und SuchIDs und stellt das Eventhandling zur Verfügung.

3.2 Funktionsbeschreibung

Initialisierung und Aufbau der Eingabemaske

Der Suchagent wird gestartet, indem der Agentenbenutzer die entsprechende URL in sein Broeser eingibt. Damit wird automatisch der `InputmaskServlet` initialisiert, der über die Methode `createSessionObjects` in der `Flowcontrol` alle für die Suche notwendigen Objekte wie Instanzen von `Input`, `StartConcept`, `T_Box`, `A_Box`, `OperatorDB`, `Results`, `Output`, `ClassifiererDB` und `Planarchive` instantiiert, und sie in das MainTable-Objekt schreibt, falls MainTable noch leer ist, d.h. die gerade erzeugte Flowkontroll ist die erste und einzige in diesem Moment. Sind alle Objekte in MainTable bereits vorhanden, werden sie ausgelesen. Die Daten in MainTable können dann auch von anderen Servlets benutzt werden.

Die `MainTable` besteht aus globalen A– und T–Box, ClassifiererDB, OperatrDB und einer Liste. In jedem Element dieser Liste werden für eine Suche aktuellen Instanzen von SearchID, ReferenceTable, Results, StartConcept, Planarchive und Input gespeichert. Dadurch wird Multiuserfähigkeit erreicht, d.h. die anderen Servlets werden vom InputmaskServlet unter Eingabe der SearchID aufgerufen, und sie suchen dann in MainTable nach „ihren“ Daten.

Die `ClassifiererDB` wird mit Hilfe von `readObject` aus der Datei geladen, die in der `Flowcontrol` angegeben wird. Die `OperatorDB`, `A_Box` und `T_Box` werden entweder als Objekt geladen oder, falls eine entsprechende Datei nicht existiert, aus einer Textdatei gelesen (vergleiche auch Abschn. 4.2 und 4.3). Falls eins von den o.g. Objekten nicht geladen werden konnte, kann die Suche nicht ausgeführt werden und wird abgebrochen. Der Agentenbenutzer bekommt eine ensprechende Fehlermeldung.

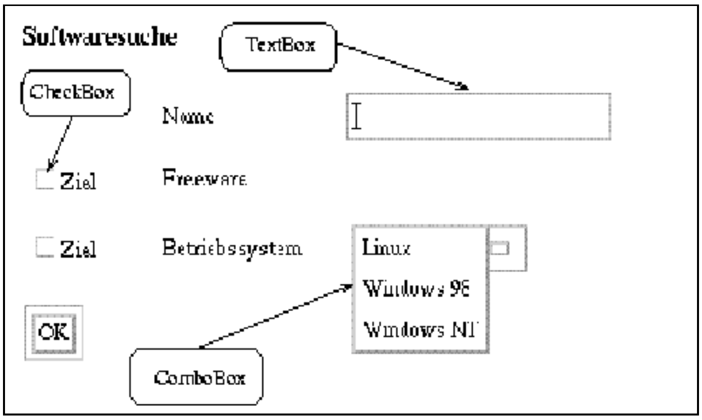
War das Laden von o.g. Objekten erfolgreich, wird die Eingabemaske generiert. Dafür wird die Methode `generateInputMask` im `StartConcept` aufgerufen. Die Methode geht alle Eingabeelemente durch und ruft in jedem Element `generateElement` auf. Diese Methode erstellt das Element zur Ausgabe im HTML-Format.

Es gibt drei Typen von Eingabeelementen :

1. TextBoxElement : Kann eine Zielcheckbox haben. Wird diese gewähl, handelt es sich bei dieser Eingabe um das Suchziel. Außerdem besteht dieser Element aus Namen und einem einfachen Texteingabefeld in dem der Benutzer den Suchstring eingibt, oder Eingaben über das Suchziel macht.

2. CheckBoxElement : Besteht aus einem Namen und einer Zielcheckbox.

3. TComboElement : Genau wie das erste Element kann dieses eine Zielcheckbox enthalten, muß aber nicht. Außerdem besteht es aus einem Namen und einer Combobox (eine Liste von auswählbaren Einträgen).



Ein Beispiel dafür, wie die Elemente in einem Browser dargestellt werden könnten!

Suchzyklus

Hat der Agentenbenutzer seine Suchanfrage in die Eingabefelder eingetragen, drückt er den OK-Button am Ende der Seite. Damit startet er `ProcessInputServlet`, das die Methode `startSearch` in dem Objekt der `Flowcontrol` startet und ein neues Fenster öffnet, in dem am Ende der Suche die Ergebnisse präsentiert werden. Für die Ausgabe ist ein Objekt der Klasse `OutputmaskServlet` verantwortlich.

In dem ersten Fenster erscheint ein Button “Zwischenergebnisse“. Wird dieser gedrückt, erscheinen die zur Zeit vorhandenen Suchergebnisse.

Nach dem Erzeugen des OutputmaskServlets wird das eine Suche eindeutig identifizierende SearchID erzeugt. Für jeden zu startenden Objekt der Klasse `Planner` wird eine PlanID und ein PlanInformation–Objekt ergeugt. Dafür wird jeweils eine Kopie der T–Box und der A–Box angelegt. In der Methode `startSearch` werden zuerst zwei Methoden aus dem `StartConcept` aufgerufen:

1. `extractGoal` markiert in der T–Box alle als Ziele markierten Benutzereingaben.

2. `markInstance` markiert in der A–Box alle dem Benutzer bekannten Merkmale des Suchobjektes. Falls eine Checkbox gewählt wurde, wird die gleichnamige Instanz in der A–Box gefunden, und mit Hilfe der `markAsConstraint` markier. Andernfalls wird eine neue Instanz in der A–Box erzeugt und markiert.

Die Systemkontrolle wird Planer eines Typs immer nur hintereinander ausführen, nicht gleichzeitig, da die Planer dann alle denselben Plan herausfänden. Planer verschiedenen Typs können und sollen gleichzeitig ausgeführt werden.

Für die `Systemkontrolle` wird intern eine Liste von Objekten der Klasse `PlnaObject` erzeugt, wobei jedes Objekt ein PlanInformation–Objekt und den damit gestarteten Planer enthält, damit die Systemkontrolle die verschiedenen Planer verwalten kann.

Suchergebnisse

Jedes einzelne Suchergebnis besteht aus einem Objekt der `NetEntity` (URL und eine Kurzbeschreibung) und einer Zahl, genauer einem Ranking, das die Güte des Ergebnisses beschreibt. Für jeden erfolgreich abgearbeiteten Gesamtplan wird eine Liste der Ergebnisse erzeugt. Sind alle erstellten Pläne ausgeführt, werden diese Listen in eine Gesamtliste – ResultsList geschrieben.

Bevor die Ergebnisse ausgegeben werden, werden die doppelt vorkommenden Einträge entfernt. Sie werden durch einen Vergleich der URLs erkannt. Danach werden die Ergebnisse gemäß ihrem Ranking sortiert, damit der Benutzer die besseren Treffer am Anfang der Seite sieht.

Werden Zwischenergebnisse verlangt, so werden die zu diesem Zeitpunkt im `ResultsList`–Objekt vorhandene Einträge sortiert und ausgegeben. Die Ausgabe wird immer in der Klasse `Output`, durch die Methode `generateOutputMask` erzeugt.

Ende der Suche

Wird das Outputservlet gestartet, so holt es alle notwendigen Objekte wie ResultsList, Output und Flowcontrol aus dem Session-Objekt. Danach wartet es bis alle Planer und Planausführungen fertig sind, und ruft dann die Methode `generateOutputMask` des `Output`–Objektes auf. Falls die Suche ohne Ergebnisse blieb, wird eine entsprechende Meldung ausgegeben. Sonst bereitet es die Ergebnisse zur Ausgabe vor (entfernt doppelte Einträge und sortiert alle Elemente), und gibt eine fertige HTML-Seite zurück. Das Servlet gibt diese Seite aus und ruft die Methode `endSearch` in `Flowcontrol`–Objekt auf. Diese Methode speichert alle Objekte, die während der Suche geändert wurden: A–Box, T–Box, Operatorendatenbank, Klassifiketorendatenbank und Planarchiv, wobei der Dateiname für das Planarchiv mit der SuchID übereinstimmen muß. Alle anderen Objekte werden ebenfalls als Objekte gespeichert, auch dann, wenn sie zuvor aus Textdateien eingelesen wurden.

Außerdem wird der Lerner gestartet. Damit ist die Suche komplett abgeschlossen.

3.3 Implementierung eines Agenten

Nachfolgend sind für jede Klasse alle Änderungen aufgelistet, die der Shellbenutzer vornehmen muß, um einen lauffähigen Agenten zu erstellen. Dabei entspricht die Abschnittüberschrift jeweils dem Namen (inkl. Pfad) der anzupassenden bzw. zu ändernden Klasse.

io.Input

Das konkrete Aussehen der Eingabemaske wird je nach Aufgabe der einzelnen Agenten variieren. Die Schnittstelle besteht defaultmäßig aus einem Formular mit Eingabefeldern, das mit Hilfe eines Java-Servlets erstellt wird. Der Shellbenutzer kann die Eigabemaske jedoch beliebig gestalten. Dazu muß zuerst die Methode `createStartConcept` überschrieben werden.

Als erstes müssen die Konzepte aus der `T_Box` geholt werden, zu denen Eingabeelemente in der Eingabemaske angezeigt werden sollen. Z.B.:

```
Concept C1 = tbox.lookup("operating_system");
Concept C2 = tbox.lookup("name");
Concept C3 = tbox.lookup("status")
```

Dann müssen die Eingabeelemente erzeugt werden. Alle Argumente werden im Konstruktor übergeben.

- `TextBoxElement`:

  ```
  TextBoxElement TE1 =
        new TextBoxElement("Name", true, C2);
  ```

  Die Argumente sind:

  1. Label – Beschriftung des Eingabefeldes
  2. hasGoalButton: `true`, falls das Element eine Zielcheckbox haben soll, sonst `false`.
  3. Konzept, wobei es sich um ein in der `T_Box` definiertes Konzepte handeln muß.

- `CheckBoxElement` :

  ```
  CheckBoxElement Ch3 =
         new CheckBoxElement(true, "Freeware", C1);
  ```

  Die Argumente:

  1. defaultValue: `true` oder `false`. Ist die variable auf `true` gesetzt, wird die Zielcheckbox markiert.
  2. Label (s.o.)
  3. Konzept (s.o.)

- `TComboBoxElement` :

  ```
  LinkedList V2 = new LinkedList(); V.add("Unstable");
  V.add("Stable"); ComboBoxElement CE1 =
         new ComboBoxElement("Linux",V, "OS", true, C1);
  ```

Die Argumente:

1. defaultValue: String, der dem Benutzer als vorgewählter Eintrag presentiert wird
2. values: Liste der ComboBox–Einträge
3. Label (s.o.)
4. hasGoalButton (s.o.)
5. Konzept (s.o.)

Danach müssen alle erzeugten Elemente in die Liste `Elements` eingetragen werden:

```
Elements.add(TE1); Elements.add(CE1); Elements.add(Ch3);
```

Jetzt kann ein `StartConcept`–Objekt erzeugt werden:

```
StartConcept Start = new StartConcept("DebianSearch", Elements);
```

„DebianSearch" ist die Überschrift der Eingabemaske.

io.StartConcept

Sie können vor oder nach der Eingabemaske beliebigen Text plazieren. Beispiel:

```
public void generateInputMask() {
  String HtmlText = "";

  inputmask = "<html><head><meta http-equiv="expires"
              content="0">"+header+"</head><body>

 ------ Hier können Sie Ihren Text einfügen. ------

  <form METHOD=GET ACTION=
            "/servlet/ProcessInputServlet"><table>";

  for (int i=0; i<elements.size(); i++) {
    Element dummy = (Element)elements.get(i); switch(dummy.typ) {
      case 1: {
      TextBoxElement TextBoxDummy = (TextBoxElement)elements.get(i);
      HtmlText = HtmlText+TextBoxDummy.generateElement(i);
      break;
    } case 2: {
      ComboBoxElement ComboBoxDummy = (ComboBoxElement)elements.get(i);
      HtmlText = HtmlText+ComboBoxDummy.generateElement(i);
      break;
    } case 3: {
      CheckBoxElement CheckBoxDummy = (CheckBoxElement)elements.get(i);
      HtmlText = HtmlText+CheckBoxDummy.generateElement(i);
      break;
    } }
  } inputmask = inputmask+HtmlText+"</table>"+
    "<INPUT TYPE=SUBMIT VALUE=OK></form>
```

```
   -----  Oder hier  ------


         </body></html>";
     }
```

In der Methode `extractGoal` werden die Ziele der Suche in der `T_Box` markiert. In der Shell werden nur die Konzepte markiert. (Vergleiche Abschnitt 8.2)

Falls Sie auch Rollen markieren möchten, müssen Sie die Methode `markAsTarget` benutzen:

```
tbox.markAsTarget(tbox.lookup("archive_url"));
tbox.markAsTarget(tbox.lookup("package"));
```

systemcontrol.Flowcontrol

Als erstes müssen Sie Dateinamen für `A_Box`, `T_Box`, `OperatorDB`, `ClassifierDatabase` und die Proxi-Werte setzen. Für die `OperatorDB` muß außerdem auch eine Textdatei definiert werden. Falls die `OperatorDB` noch nicht existiert, wird sie aus dieser Textdatei ausgelesen. Sie können die Dateinamen aus einer Konfigurationsdatei einlesen. Hier ist ein Ausschnitt aus einer Konfigurationsdatei:

```
Working Directory = /home/pg343/malzahn

# relative oder absolute Pfade der zu ladenden Dateien

A_Box=sourcen/integration-shell/DebianSuchagent.abox
T_Box=sourcen/integration-shell/DebianSuchagent.tbox
OperatorDBDate=sourcen/integration-shell/OperatorDB.opdb
OperatorDBObject=sourcen/integration-shell/OperatorDB.opdbObject
ClassDatabase=/home/pg343/share/DebianSuchagent/DebianSuchagent.cldb
```

So kann in der Methode `createSessionObjects` eine Zeile aus der Konfigurationsdatei eingelesen werden:

```
if (line.startsWith("Working Directory"))
  {
    wd=(line.substring
          (line.indexOf("=")+1,line.length())).trim();
    LogService.log(100,this,
      "current Working Directory set to:"+wd);
  }
```

systemcontrol.event.ShellEventAdapter

Die Eventhändling wird im nächsten Kapitel ausführlich beschrieben. Sie können die maximale Anzahl der Instanzen pro Operatorvorbedingungsprädikat bestimmen, die bei der Suche berücksichtigt werden. Momentan befindet sich die Variable `MaxInstance` in der Konfigdatei, und ist auf

10 gesetzt. (siehe Abschnitt II D.1) Diese Beschränkung der Belegungskonbinationen dient der Reduzierung der Laufzeit für die Suche.

Wird das Event `planExecuteFinishedEvent` ausgelöst, kann der Agentenbauer frei entscheiden, ob derselbe Planner oder ein anderer Plannertyp den nächsten Teilplan erstellen soll. Im ersten Fall soll der Programmcode nicht geändert werden. Sonst muß die Zeile

```
PO.getPlan().myResume();
```

durch

```
String Class = (PO.getPlan().getClass()).toString();
PO.getPlan().myDestroy();
if (Class == PlannerTypA) {
   PlannerTypeC plan1 = new PlannerTypeC(planInfo);
   PO.setPlan(plan1);
   }
else {
   PlannerTypeA plan1 = new PlannerTypeA(planInfo);
   PO.setPlan(plan1);
   }
plan1.start();
```

o. ä. ersetzt werden.

Andere Eingabemasken

Möchten Sie statt unserer Benutzereingabeschnittstelle eine andere benutzen, müssen Sie folgendes beachten.

Ihre Schnittstelle sollte eine Instanz der Klasse `Flowcontrol` erzeugen, die Methode `createSessionObjects` starten z.B.

```
Flowcontrol flow = new Flowcontrol(); Object objs[] =
flow.createSessionObjects();
```

Dabei müssen Sie beachten, daß in dieser Methode Objekte der Klassen `Input`, `Output` und `StartConcept` erzeugt werden, und ein Aufruf von `createStartConcept` erfolgt. Ihre Klassen müssen d ieselbe Funktionalität wie die, von uns zu Verfügung gestellten aufzuweisen.

Danach müssen Sie die Eingabemaske generieren.

Hat der Agentenbenutzer die Maske ausgefült und abgeschickt, müssen die von ihm gemachten Eingaben in der A–Box mit `markAsConstraint` und die Ziele in der T–Box mit `markAsTarget` markiert werden. Außerdem muß die Suche mit einem Aufruf von `startSearch` in einem Objekt der Kalsse `Flowcontrol` gestartet werden. Wichtig ist, daß auch diese Methode dieselbe Instanz von `StartConcept` benutzt, so daß diese Methode evtl. zum Teil geändert werden muß. Nachdem die Suche beendet wurde, muß die Ausgabemaske generiert und ausgegeben werden.

Kapitel 4

Eventhandling

Die Menschen sind da, um einander zu helfen, und wenn man eines Menschen Hülfe in rechten Dingen nötig hat, so muß man ihn dafür ansprechen. Das ist der Welt Brauch und heißt noch lange nicht betteln.

Jeremias Gotthelf

Planer, Planausführer und die Systemkontrolle kommunizieren mit einander mit Hilfe von Events. Hat ein `Planner`–Objekt einen Teilplan erstellt, so löst er ein Event aus. Folgende Events sind möglich:

1. `planReadyFinishedEvent` : Der Planer hat einen letzten Teilplan erstellt. Nach dessen erfolgreicher Ausführen müssen die Ergebnisse vorliegen.
2. `planReadyUnfinishedEvent` : Nach der Ausführung des erstellten Teilplans muß ein `Planner`–Objekt mit dem vorhandenen Wissen weiterplanen.
3. `planReadyAborted` : Der Planer konnte mit den vom Benutzer eingegebenen Daten keinen Plan erstellen. Die Suche wird abgebrochen. Der Planer wird angehalten und gelöscht.

Wurde ein Teilplan erstellt, also eines der ersten beiden Events ausgelöst, wird der Planer angehalten und mit dem aktuellen `PlanInformation`–Objekt eine Planausführung gestartet.

Die Planausführung gibt das Ende der Ausführung durch Events bekannt. Das `planExecuteFinishedEvent` gibt an, daß die Planausführung erfolgreich verlaufen ist. Falls der ausgeführte Teilplan nicht der letzte Teilplan des Gesamtplans war, wird ein Planer mit dem `PlanInformation`–Objekt angestoßen. Der gestartete Planer kann derselbe sein, der den vorherigen Teilplan erstellt hat, oder ein anderer Planertyp. Die Entscheidung liegt bei dem Shellbenutzer.

Sonst wird der Planer, der diesen Teilplan erstellt hat, zerstört, die gewonnenen Ergebnisse werden in eine Liste geschrieben. Das `PlanInformation`–Objekt wird in das Planarchiv eingetragen. Außerdem wird überprüft, ob alle Planer bereits dei Suche komplett abgeschlossen haben. In diesen Fall gilt die Suche als beendet.

Das `planExecuteFinishedBackEvent` wird nur dann aufgerufen, wenn der ausgeführte Teilplan nicht der letzte war, und es ist unbedingt notwendig, daß derselbe Planer auch weiterplant.

Das `planExecuteAbortedEvent` gibt an, daß die Planausführung aus irgendeinem Grund scheiterte. Dieser Planer wird als fertig markiert d.h. er hat diese Suche abgeschlossen, und wird nicht mehr gestartet. Falls alle anderen Planer auch markiet sind, wird die Suche beendet.

Ein Planer oder eine Planausführung könnten außerdem auch ein `planExecuteFinishedEvent` auslösen, falls ein letzter Teilplan erstellt und ausgeführt wurde. Der Planer, der diesen teilplan erstellt hat, wird dann ebenfalls zerstört, und die vorhandenen Ergebnisse werden in die Liste der Suchergebnisse eingetragen.

Will ein Planer oder eine Planausführung ein Event auslösen, muß er zuerst ein Event erzeugen:

```
 ShellEvent e = new ShellEvent(this, getPlanInfo());
```

Dann muß er einen ensprechenden Listener registrieren:

```
ShellEventListener sel =
     (ShellEventListener)Flowcontrol.ShellEventListeners.elementAt(0);
```

Jetzt kann das Eventauslösende Objekt beliebige Events erzeugen:

```
sel.planReadyAborted(e);
```

Kapitel 5

T– und A–Box

Wissen ohne Ordnung ist Hausrat auf einem Leiterwagen

Jakob Lorenz

5.1 Überblick und Definition

Für die Wissensrepräsentation wird das Modell einer T- und A-Box verwendet. Die T–Box ist der terminologisch Teil, in der die Begriffe definiert und in eine Begriffsstruktur eingeordnet werden. Die T–Box stellt das konzeptionelle Weltwissen des Suchagenten dar und muss vor der Initialisierung modelliert werden. Deshalb ist der Inhalt der T–Box ist entscheidend für die Sucherfolge des Agenten.

Die A–Box ist der assertionale Teil der Wissensrepräsentation, d.h. in der A–Box werden die Individuen oder Instanzen und die tatsächlich bestehende Rollen gespeichert. Auch die A–Box muß vor der ersten Suche mit einigen Instanzen gefüllt werden, damit der Planer einen Suchplan entwickeln kann.

Termini der T–Box:

Def.: Ein Konzept repräsentiert eine Menge von Instanzen.

Def.: Ein Konzept K heißt primitives Konzept , wenn seine Menge von Instanzen eine Untermenge der Schnittmenge seiner Oberkonzepte $K_1, ..., K_n$ ist. Sei U die Instanzenmenge von K und $O_1, ...O_n$ die Mengen von $K_1, ..., K_n$, dann gilt $U \subseteq O_1 \cap ... \cap O_n$.

Def.: Ein Konzept heißt definiertes Konzept, wenn seine Menge von Instanzen gleich der Schnittmenge seiner Oberkonzepte $K_1, ..., K_n$ ist. Sei U die Instanzenmenge von K und $O_1, ...O_n$ die Mengen von $K_1, ..., K_n$, dann gilt $U \subset O_1 \cap ... \cap O_n$.

Def.: Eine Rolle ist eine binäre Relation zwischen Konzepten.

Def.: Eine *isA*–Beziehungen zwischen Konzepten ist keine Rolle im oben genannten Sinne, sondern stellt die Konzepthierachie dar.

Termini der A–Box:

Def.: Eine Instanz ist eine Entität eines Konzeptes. Instanzen können gleichzeitig zu verschiedenen Konzepten gehören.

Def.: Eine konkrete Rolle ist eine tatsächlich bestehende Rolle zwischen zwei Instanzen und wird häufig auch als Rolleninstanz bezeichnet.

5.2 T–Box

Die T–Box wird als Graph dargestellt, wobei die Knoten die Konzepte bilden und die Kanten die möglichen Rollen zwischen den Objekten. Ausserdem gibt es eine ausgezeichnete Kantenmenge, die die *isA*–Beziehungen zwischen den Konzepten darstellt und die Hierachie zwischen Oberkonzepten und Unterkonzepten repräsentieren. Sie alleine bilden mit den Konzepten einen Baum mit `anything` als Wurzel, unter welches jedes Konzept subsummiert wird. Es existiert ein Subsummationstest, der die Konzepte a und b darauf überprüft, ob a b subsummiert. Wenn das der Fall ist, ist b spezieller als a und a ist das Oberkonzept von b. Dann ist b ein primitives Konzept.
Wird das Konzept b von den Konzepten $a_1, ..., a_n$ subsumiert, wobei die Konzepte $a_1, ..., a_n$ sich nicht subsummieren und b genau den Konzepten $a_1, ..., a_n$ entspricht, ist b ein definiertes Konzept. Zusätzlich subsummiert sich jedes Konzept auch selbst.
Die Rollen in der T–Box müssen eindeutige Namen haben, weil sowohl die Startinstanzen, als auch die Zielinstanz zu mehren Konzepen gehören können, und somit nur der Rollennamen die Eindeutigkeit garantiert.

Außerdem werden in der T-Box Konzepte als Ziele der Suche markiert. Instanzen, die unter dieses Konzepte fallen, bilden die Suchergebnisse. Aus diesen Anforderungen folgt auch, daß zu jeder Suche eine eigene T–Box erzeugt werden muß.

Beispiel

Ein Ausschnitt einer T–Box für einen Software–Suchagenten könnte aussehen wie in Abbildung II 5.1:

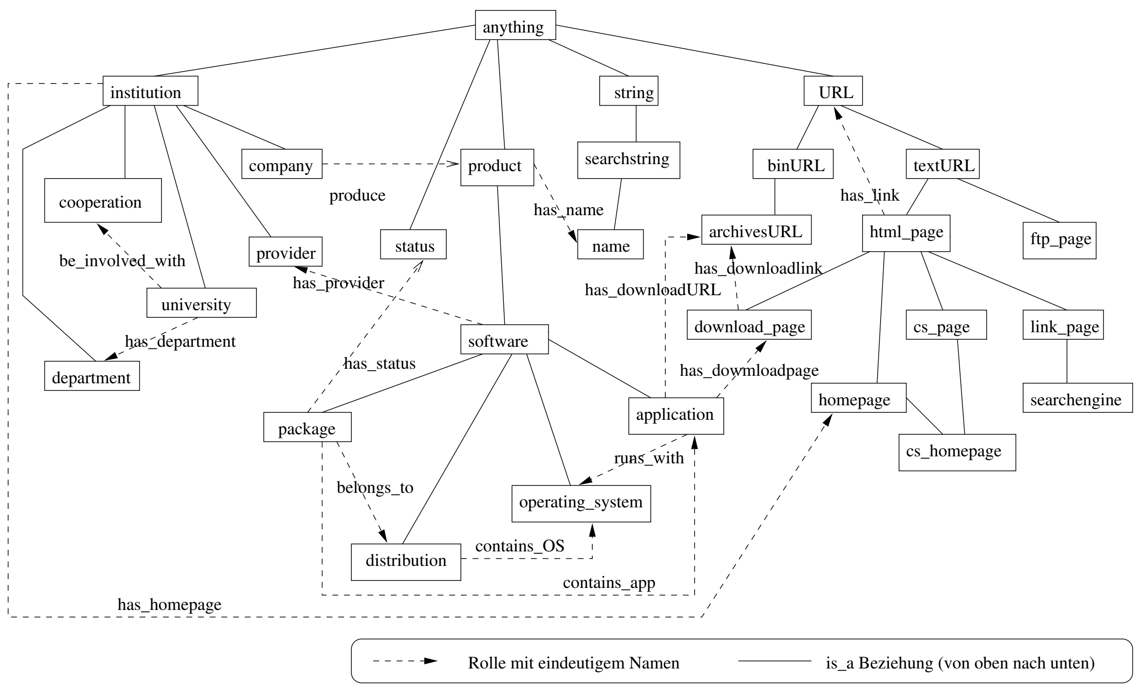


Abbildung 5.1: Mögliches Aussehen der T–Box für einen Softwaresuch–Agenten

Dieses kleine Beispiel zeigt, wie umfangreich eine T–Box, und wie mühsam eine graphische Darstellung werden kann. Dennoch ist es ratsam, eine solche Darstellung zu erstellen, weil so am schnellsten deutlich wird, wo Modellierungsfehler sind, z.B. wenn ein Konzept keinerlei *isA*–Beziehungen hat, nicht einmal zu `anything`, oder wenn es durch die modellierten Rollen möglich wird, unsinnige konkrete Rollen zu den Instanzen zu bilden, was nicht erwünscht ist.

Die drei linken Äste des Hierachiebaums ist abhängig vom jeweiligen Suchgebiet, auf den Rest wird wohl kein Suchagent, der im Internet suchen soll, verzichten können.

An diesem Beispiel kann auch noch einmal der Unterschied zwischen einem primitiven und einem definierten Konzept deutlich gemacht werden. *`homepage`* ist ein primitiven Konzept, weil *`homepage`* zwar ein Unterkonzept von *`html_page`* ist, also jede *`homepage`* ist auch eine *`html_page`*, aber nicht jede *`html_page`* ist auch eine *`homepage`*. Dagegen gehört eine Instanz nur dann zum Konzept *`cs_homepage`*, wenn sie sowohl eine *`homepage`* und als auch eine *`cs_page`* ist.

Dateiformat der T–Box

Um dem Shell–Benutzer eine etwas komfortabelere Möglichkeit zu bieten, die T–Box anzulegen, ist es sinnvoll, die Dateien in einer lesbaren Form anzulegen. Deshalb kann der Inhalt der T–Box mit Hilfe einer Textdatei eingegeben werden. Das Dateiformat wird zeilenweise verarbeitet, so daß in jeder Zeile genau ein Kommando stehen darf. Zudem muß folgend Syntax beachtet werden:

- Groß- und Kleinschreibung wird nicht unterschieden.
- Kommentare werden mit # eingeleitet. Alles was in der Zeile nach # steht, wird ignoriert.
- Zwischen den Bezeichnern muß immer mindestens ein Leerzeichen stehen.
- Primitive Konzepte werden in der Form $A :< B$ bzw. $A :< B$ *and* C definiert. Dabei ist das Konzept A das primitive Konzept, welches von dem Konzept B bzw. von den Konzepten B und C subsummiert wird.
- Definierte Konzepte werden in der Form $A := B$ bzw. $A := B$ *and* C definiert. Dabei ist das Konzept A das definierte Konzept, welches von dem Konzept B bzw. von den Konzepten B und C subsummiert wird.
- Das alle Konzepte subsummiernde Oberkonzept heißt *anything* und das inverse kein Konzept subsummiernde Konzept heißt *nothing*.
- Rollen werden in der Form $R:\ A => B$ definiert, mit R ist die zu definierende Rolle, A das Konzept, in dem die Rolle definiert ist, und B das Konzept, auf das die Rolle verweist.
- Fehlerhafte Zeilen werden beim Laden mit einer Fehlermeldung aufgelistet und bei der weiteren Abarbeitung ignoriert.

Zurück zum Beispiel in Abb. II 5.1

Entsprechend zu der Modellierung sehen Einträge in der T–Box–Textdatei dann so aus:

```
# Konzepte die Computer betreffen

# primitive Konzepte
operating_system :< software
application :< software
cs_page :< html_page
homepage :< html_page

# definiertes Konzept
cs_homepage := cs_page and homepage
```

```
# Rollen
runs_with : application => operating_system
contains_OS : distribution => operating_system
```

5.3 A–Box

Die A–Box ist eine Liste, die als Elemente die Instanzen enthält. Es können nur solche Instanzen in die A–Box geschrieben werden, die zu einem Konzept aus der T–Box gehören. Eine Instanz kann mehr als einem Konzept zugeordnet werden, wobei die Konzepte nicht in Oberkonzept–Unterkonzept–Relation zueinander stehen müssen. Jede Instanz hat eine Liste von Konzepten, denen sie zugeordnet wird, und zudem hängt an jedem Konzept ein Wahrscheinlichkeitswert, der angibt, mit welcher Wahrscheinlichkeit die Instanz unter dieses Konzept fällt. Der Wahrscheinlichkeitswert ist ein *double* im Intervall von [0,1].

Außerdem hat jede Instanz eine Liste mit den konkreten Rollen, die zu dieser Instanz existieren. Auch an den konkreten Rollen hängt ein Wahrscheinlichkeitswert im Intervall von [0,1], der angibt, mit welcher Wahrscheinlichkeit die Rolle der Instanz zugeordnet werden kann. Es können ähnlich wie bei den Konzepten nur dann konkrete Rollen angelegt werden, wenn es in der T–Box eine Rolle mit diesem Namen gibt und sowohl die Startinstanz, als auch die Zielinstanz in ihrer Konzeptliste die Konzepte haben, zwischen denen die Rolle in der T–Box definiert ist.

Neben der eindeutigen Id–Nummer und dem Namen der Instanz, der nicht eindeutig sein muß, kann ein dazugehörender Inhalt vom Typ ***NetResult*** gespeichert werden. Zusätzlich hat jede Instanz noch weitere Attribute, wie z.B. `Instance.date`, das das Eintragsdatum festhält, oder `Instance.counter`, der angibt, wie oft eine Instanz zur Planerstellung benutzt wurde, und noch einige andere, die für den Planer oder den Lerner wichtig sind.
Genau wie bei der T–Box wird es mehrere A–Boxen gleichzeitig geben, eine globale, die über alle Suchen persistent ist und die vor der Initialisierung des Suchagenten mit Inhalt gefüllt werden muß. Für jede Suchanfrage wird eine Kopie der globale A–Box erzeugt, die sogenannte suchglobale A–Box, in der die Instanzen, die durch die Anfrage hinzu gekommen sind markiert werden. Für jeden Planer wird eine Kopie der suchglobale A–Box angelegt, die durch einen Planausführer mit neu dazukommenden Instanzen angefüllt wird. Ist eine Suche abgeschlossen, überträgt anschliessend der Instanzenlerner die neu hinzugekommenden Instanzen, soweit es sinnvoll erscheint, in die globale A-Box.

Dateiformat der A–Box–Datei

Wie bei der T–Box, ist es auch bei der A–Box möglich, die globale A–Box mit Hilfe einer Textdatei zu füllen. Die Textdatei wird dann durch den Aufruf von `A_Box.load` in eine leere A–Box eingelesen.
Da die Textdatei zeilenweise verarbeitet wird, sind beim Erstellen einige Regeln zu beachten:

1. Kommentare werden mit # eingeleitet. Alles was in einer Zeile nach einem # steht, wird ignoriert.

2. Jede einzelne Instanz wird durch <Instance> eingeleitet und durch <\Instance> beendet. Die einzelnen Komponenten, wie z.B. der Namen der Instanz werden danach zeilenweise eingegeben.

3. Ähnliches gilt für die Aufzählung von Rollen, sie beginnt mit `<roles>`, dann kommen zeilenweise die einzelnen Rollen und endet mit `<\roles>`. Auch die Konzepte beginnen mit `<concepts>`, dann kommen zeilenweise die Konzepte, evtl. der Wahrscheinlichkeitswert in eckigen Klammern. Die Aufzählung der Konzepte wird mit `<\concepts>` abgeschlossen.

Zurück zum Beispiel in Abb. II 5.1

Instanzen der A–Box könnten dann z.B. folgende sein:

```
# im Bereich Betriebssysteme
<Instance>
name: linux
<concepts>
operating_system [1]
<\concepts>
<\Instance>

<Instance>
name: debian # Linux-distributionen
<concepts>
distribution
<\concepts>
<roles>
debian contains_OS linux [0.8]
<\roles>
<\Instance>
```

Wird die A–Box mittels der Textdatei aufgefüllt, muß der Instanzenname eindeutig sein, weil die Id–Nummer erst nach dem Aufruf von `A_Box.load` vergeben wird und die Instanz erst ab diesem Zeitpunkt durch ihre Id identifiziert wird. Der Shellbenutzer muß beim Eintragen der Instanzen darauf achten, daß das oder die Konzepte in der T-Box vorhanden sind und daß in der T-Box die entsprechenden Rollen existieren. Wird bei den Konzepten und konkreten Rollen kein Wahrscheinlichkeitswert angegeben, wird er beim laden automatisch auf 1 gesetzt. Sind Sie sich aber nicht sicher, ob ein Konzept oder eine konkrete Rolle tatsächlich zutrifft, können Sie auch einen beliebegen anderen *double*–Wert im Intervall von [0,1] in eckigen Klammern an die konkrete Rolle oder das Konzept schreiben.

Die Instanzattribute `Instance.counter`, `Instance.date`, `Instance.delete`, usw. werden beim Erzeugen der Instanz automatisch gesetzt. Wird die A-Box erneut gespeichert, werden `Instance.id`, `Instance.date` und `Instance.counter` mitgespeichert. Dann ist es möglich, daß doppelte oder kein Instanzennamen existiert, die Instanzen können aber anhand ihrer Ids eindeutig identifiziert werden.

Instanzen, die weder einen Namen noch eine Id haben, werden beim Laden nicht erzeugt. Das gilt auch für Instanzen, bei denen kein Konzept angegeben ist, oder das angegebene Konzept nicht in der T-Box existiert. Dagegen werden Instanzen, deren Rollen in der T-Box nicht vorhanden sind, erzeugt, die konkreten Rollen aber nicht angelegt. In diesen Fällen gibt es dann eine Fehlermeldung.

Die Funktion `A_Box.save` sollte nicht zum Speichern der globalen A–Box benutzt werden, weil der `Instance.contens` nicht mit in die Textdatei gespeichert wird.Die Funktion `A_Box.save` dient lediglich dazu, die A–Box wieder in die lesbare Form der Textdatei zu überführen. Möchte man später neue Instanzen in die globale A–Box schreiben, kann man das auch wieder mit Hilfe einer neuen Textdatei tun. Die neuen Instanzen werden dann durch `theGlobalA_Box.load` zu den schon existierenden Instanzen dazu geladen. Sie müssen bei dieser neuen Textdatei nicht darauf achten, ob Sie Instanzennamen vergeben, die in `theGlobalA_Box.load` schon vorkommen, nur innerhalb der neuen Textdatei dürfen Instanzennamen nicht doppelt vergeben werden.

Eigene Änderungen der Implementierung

Es empfiehlt sich nicht, die Wissensrepräsentation zu verändern, da sie ein zentraler Bestandteil der Shell ist und eng mit den Plannern, den Planausführern und der Systemkontrolle zusammenarbeitet. Sollten Sie das denoch wünschen, müssen Sie die Schnittstellen, die im Java–Doc beschrieben sind, einhalten.

Wollen Sie die Syntax der T–Box–Textdatei, bzw. der A–Box–Textdatei ändern, müssen Sie die Methoden `T_Box.load`, `T_Box.save`, `A_Box.load` und `A_Box.save` entsprechend anpassen.

Kapitel 6

Planarchiv

> Ich habe mir die Zeitungen vom vorigen Jahr binden lassen. Es ist unbeschreiblich, was für eine Lektüre dieses ist: 50 Teile falsche Hoffnung, 47 Teile falsche Prophezeiungen und 3 Teile Wahrheit. Diese Lektüre hat bei mir die Zeitungen von diesem Jahr sehr herabgesetzt; denn ich denke: Was diese sind, das waren jene auch.
>
> *Georg Christoph Lichtenberg*

In das Planarchiv schreibt die Systemkontrolle die Planinfo–Objekte, die zusammen einen Suchverlauf repräsentieren. Diese Pläne werden in einer globalen Datei gespeichert und werden von den Lernern benutzt.

Kapitel 7

Operatoren

Zum Erfolg gibt es keinen Lift, man muß die Treppe benützen.

Emil Oesch

7.1 Operatoren im Überblick

Wenn Sie etwas erreichen wollen, müssen Sie normalerweise auch etwas dafür tun. Um etwas machen zu können, müssen meistens gewisse Vorbedingungen erfüllt sein. Ist schließlich Ihre Arbeit nicht ganz für die Katz', dann kommt auch noch etwas Produktives dabei heraus.

Wollen Sie z. B. einen Bilderrahmen mit einem Nagel an einer Wand befestigen, so geht das nur, wenn Sie auch einen herumliegenden Bilderrahmen, einen herumliegenden Nagel, eine freie Wand und natürlich einen Hammer haben. Geht alles glatt, so haben Sie anschließend keinem herumliegenden Bilderrahmen mehr, keinen herumliegenden Nagel und auch keine freie Wand. Geblieben ist Ihnen nur der Hammer und die Information, daß der Rahmen jetzt mit dem Nagel an der Wand befestigt ist. Bei gleichen Anfangsbedingung kann es aber auch sein, daß Sie nach der Ausführung Ihres Vorhabens eine immer noch freie Wand, einen herumliegenden Bilderrahmen, einen krummen, herumliegenden Nagel und einen dicken Daumen haben. Den Hammer haben Sie immer noch.

An diesem kleinen Beispiel ist deutlich zu sehen, daß die Ausführung einer Aktion nach folgendem Schema abläuft:

1. Was brauche ich?
2. Habe ich alles?
3. Ausführen!
4. Was habe ich jetzt?

Wenn Sie eine Operation (eine (oft) kompliziertere Aktion oder Handlung) machen sollen, so können Sie normalerweise sagen, was Sie alles brauchen und was Sie dann mit den einzelnen Sachen machen würden. Das ist völlig unabhängig davon, ob Sie das dann auch wirklich tun. Genausogut können Sie im vorhinein sagen, was als Ergebnis herauskommen könnte. Das eine Operation durchaus mehr als ein Ergebnis haben kann, haben wir in obigem Beispiel gesehen!

Was Sie nicht allgemein formulieren können ist, ob nun in dem Moment, in dem Sie die Operation ausführen wollen, auch alle Vorbedingungen erfüllt sind. Genausowenig können Sie, auch wenn alle Vorbedingungen erfüllt sind, wissen, was nachher bei Operationen mit mehreren möglichen Ausgängen herauskommt.

Die Beschreibung dessen, was eine Operation an Voraussetzungen hat, was mit den notwendigen Dingen dann gemacht würde und welche Ergebnisse dabei herauskommen könnten, steckt in der Klasse `Operator`. Leider ist diese Klasse abstrakt, so daß Sie, lieber Shellbenutzer, erst einmal Operatoren bauen müssen, die auf dem Weg zur Lösung genau Ihres Problems Ihnen die Steine aus dem Weg räumen können.

Anders ausgedrückt: wenn Sie einen eigenen Agenten schreiben möchten, so sind die Operatoren die Stelle, an denen Ihnen BotIshelly die wenigste Arbeit abnehmen kann, und Sie am meisten kaputt machen können!

Gehen wir es also an! ;-)

7.2 Prädikate und ihre Argumente

Prädikate sind Aussagen über irgendetwas. Das, worüber Aussagen getroffen werden, nennen wir auch die Argumente eines Prädikats. Setzen wir zwei Argumente über ein Prädikat in Beziehung, so sprechen wir auch von einer Rolle zwischen den beiden Argumenten. Aus der Mathematik kennen Sie die Rolle „kleiner gleich". Prinzipiell kann man eine Aussage treffen wie $a < b, \quad a, b \in \mathbf{N}$, mehr programmiersprachlich aufgeschrieben als `Less (a : Nat, b : Nat)`. Dabei bezeichnen wir `Less` als den Namen des Prädikats, `a` und `b` als die Namen von Prädikatargumenten und `Nat` als Konzept oder Typ, auf den die möglichen Prädikatargumente eingeschränkt sind. Sie können z. B. auch notieren `Prime (a : Nat)`, wenn Sie ausdrücken wollen, daß es sich bei `a` um eine Primzahl handelt, oder auch `Nat (a : Nat)`, um `a` als natürliche Zahl auszuzeichnen.

Dererlei Prädikate können Sie in der Praxis nur anwenden, wenn Sie die Variablen auch mit Instanzen belegen können. Ein Prädikat, dessen Argumente vollständig mit Instanzen belegt sind, nennt man instantiiert. So sind z. B. `Less (a=1: Nat, b=3: Nat)` oder auch `Less (a=3: Nat, b=1: Nat)` vollständig instantiiert, obwohl letztere Aussage natürlich falsch ist.

Im vorangehenden Beispiel bezeichnen wir `Less` als den Namen des Prädikats (oder auch der Rolle, da das Prädikat zweistellig ist), `a` und `b` als Namen der Prädikatargumente, die Zahlen 1 und 2 als Instanzen der Prädikatargumente und die beiden `Nat` als Konzepte der Prädikatargumente.

BotIshelly stellt entsprechend die Klassen `PredArgument` und `Predicate` im Paket `operators` als Prädikatargumente und Prädikate zur Verfügung. Bei der Erzeugung einer Java–Instanz ***`PredArgument`*** ist darauf zu achten, daß das im Konstruktor

```
public PredArgument(Concept type, String name)
```

übergebene Konzept aus der T–Box stammen muß, die gerade benutzt wird. Sie dürfen sich die Java–Instanz ***`Nat`*** eines Konzeptes `Nat` also nie selber durch einen Aufruf von `new` erzeugen, sondern müssen immer mit Hilfe von z. B. ***`T_Box`***`.lookup("Nat")` die von der T–Box angebotene Instanz ***`Nat`*** nehmen.

Ein Prädikat läßt sich durch Angabe seiner Stelligkeit (engl. *arity*) und des Prädikatnamens über den Konstruktor

```
public Predicate(int predArity, String predName)
```

erzeugen. Hier liegt es in Ihrer Verantwortung, darauf zu achten, daß sie nur Namen eintragen, die auch so als Namen von Konzepten oder Rollen in der T–Box vorkommen!

Als kleines Beispiel sei hier noch das Erzeugen einer Rolle für die kleiner–Beziehung aufgezeigt:

```
Predicate predicate = new Predicate (2, "less");
Concept natural = tbox.lookup ("natural");
predicate.setPredArg (new PredArg (natural, "a"), 0);
predicate.setPredArg (new PredArg (natural, "b"), 1);
```

Bitte beachten Sie, daß zu diesem Zeitpunkt noch keine Instanzen in den PredArgs belegt oder gar erzeugt sind!

7.3 Vorbedingungen und Nachbedingungen

`[Nagel (nagel : Nagel), Wand (wand : Wand), Rahmen (rahmen : Rahmen), Hammer (hammer : Hammer)]` wäre eine passende Vorbedingung für einen Operator, der einen Bilderrahmen an einer Wand befestigen kann, wobei die Liste so zu lesen ist, daß alle Prädikate erfüllt sein müssen (Konjunktion). Denkbar sind natürlich auch andere boolesche Verknüpfungen, wie Disjunktionen oder Negationen. Aufgrund der Komplexheit läßt BotIshelly aber nur Konjunktionen von Prädikaten als Vor– und Nachbedingungen zu.

Obwohl sich Vor– und Nachbedingungen recht ähnlich sind, gibt es doch einige Unterschiede. Z. B. gibt es genau eine Vorbedingung für einen Operator. Nehmen wir an, Sie haben einen Operator mit einem Stein als Vorbedingung, der damit eine Scheibe einschmeißen kann. Wollen Sie den Operator erweitern, so daß Sie jetzt auch mit einem Hammer eine Scheibe einschmeißen können, so können sie als Vorbedingung [Hammer oder Stein] *nicht* formulieren. Sie können das Problem auf zwei Arten angehen:

1. Sie schreiben einfach einen neuen Operator, indem sie den mit dem Stein kopieren und als Vorbedingung jetzt Hammer eintragen.
2. Sie ändern die T–Box so ab, daß Sie ein gemeinsames Oberkonzept für Hammer und Stein (z. B. SchwererGegenstand) einfügen. Entsprechend würde dann die Vorbedingung für den Operator jetzt `[SchwererGegenstand (schwererGegenstand : SchwererGegenstand)]` lauten.

Am Ausgang des Bildaufhängens haben wir gesehen, daß es mehr als eine Nachbedingung geben kann. Zusätzlich dazu, daß nur Konjunktionen erlaubt sind, schränkt BotIshelly hier weiter dadurch ein, daß die Nachbedingungen nur exklusiv–oder miteinander verknüpft sein dürfen. Wenn Sie schreiben wollen, daß der Daumen blutet, das Bild hängt oder beides passiert ist, so müssen Sie drei Nachbedingungen (vereinfacht)

1. `[BildAnWand (wand : Wand, bild : Bild)]`
2. `[DaumenBlutet (daumen : Daumen)]`
3. `[BildAnWand (wand : Wand, bild : Bild),`
   `DaumenBlutet (daumen : Daumen)]`

schreiben. Sollte die erste Nachbedingung erfüllt sein, so können Sie daraus nicht schließen, daß der Daumen nicht blutet. Sie wissen nur, daß der Daumen aufgrund der Ausführung des Bildaufhäng–Operators nicht blutet. Falls das Prädikat `DaumenBlutet` vor der Anwendung des Operators schon erfüllt war, so ist es danach natürlich immer noch erfüllt.

Ein Operator kann bei der Ausführung dreierlei Sachen mit Prädikaten und Rollen machen:

1. Ein `Predicate` der Vorbedingung wird nicht verändert.

2. Ein `Predicate` der Vorbedingung wird gelöscht.

3. Ein `Predicate` wird neu erzeugt.

Aus diesem Grund sind einzelne Nachbedingungen nicht nur Listen von Prädikaten, sondern bestehen aus zwei Listen, einer Add– und einer Deleteliste. In der Addliste einer Nachbedingung sind alle diejenigen Prädikate und Rollen eingetragen, die durch den Operator neu erzeugt wurden. In der Deleteliste stehen dann natürlich diejenigen, die durch den Operator gelöscht wurden. Da der Operator sein Wissen *nur* aus den Vorbedingung schöpfen kann, dürfen in den Deletelisten der Nachbedingungen verständlicherweise auch nur Prädikate oder Rollen auftauchen, die schon in der Vorbedingung standen. Implizit vorausgesetzt wird, daß ein `Predicate` der Vorbedingung, daß nicht in der Deleteliste einer Nachbedingung auftaucht, beim Erreichen genau dieser Nachbedingung auch nicht verändert worden ist.

BotIshelly bietet für Nachbedingungen die Klasse `Postcondition` im Paket `operators` an, die über den leeren Konstruktor `Postcondition()` erzeugt wird. Die beiden Methoden `addAddListPred (Predicate p)` und `addDeleteListPred (Predicate p)` dienen zum Manipulieren der Add– und Deleteliste der einzelnen Nachbedingung. Die Add– und Deletelisten müssen natürlich disjunkt sein, da es keinen Sinn macht, als Ergebnis eines Operators ein Prädikat gleichzeitig zu erzeugen und zu löschen.

Wenn Sie in den Vor– und Nachbedingungen eines Operators die Namen für die Prädikatargumente vergeben, müssen Sie einige Einschränkungen beachten, die sich aus der Implementierung der Planausführung ergeben: Einstellige Prädikate müssen zwingend wie das Konzept des zugehörigen Prädikatargumentes heißen. Daraus folgt, daß es zu jedem Konzept auch nur genau einen Prädikatnamen geben kann. Bei zweistelligen Prädikaten sind Sie in der Wahl des Namens allerdings frei. Z. B.:

```
[Nat(nat1: Nat), Less (nat2: Nat, prime: Prime)]
```

Eine weitere Einschränkung ist, daß in einem Prädikat der Vorbedingung die Namen der Prädikatargumente nicht schon *alle* links von diesem vorgekommen sein dürfen. Denn eine Vorbedingung der Art

```
[Nat(nat1: Nat), Nat(nat2: Nat), Less (nat1: Nat, nat2: Nat)]
```

in der die Prädikatargumentnamen `nat1` und `nat2` von `Less` schon beide links von `Less` auftauchen, sorgt bei der Ausführung dafür, daß der Planer, der von links nach rechts die Vorbedingung zu belegen versucht, `Less(nat1, nat2)` schon für belegt hält. Dann wird aber durch den Planausführer nicht mehr überprüft, ob das Prädikat `Less` für `nat1` und `nat2` überhaupt Gültigkeit hat! Es sei an dieser Stelle noch einmal darauf hingewiesen, daß diese Einschränkungen sich aus unserer Implementierung des Planausführers ergeben.

Erlaubt dagegen ist

```
[Nat(nat1: Nat), Less (nat1: Nat, nat2: Nat)]
```

da hier beim Belegen von `Less` zwar `nat1` schon durch das linksstehende Prädikat belegt wurde, beim Belegen von `nat2` durch den Planausführer aber eine Überprüfung der Gültigkeit von `Less` stattfindet. *Ausreichend* und auch zu bevorzugen ist an dieser Stelle aber eine Vorbedingung der Form

```
[Less (nat1: Nat, nat2: Nat)]
```

da der Planausführer bei der Belegung von `nat1` und `nat2` natürlich nur Instanzen auswählt, die auch unter das Konzept `Nat` passen und damit per se das Prädikat `Nat` erfüllen.

Aus obigen Einschränkungen folgt, daß Sie z. B. für einen Operator, der als Vorbedingung $a < b$ und $a < c$ und $b < c$ fordert, diese nur schreiben können als

```
[Less (natA: Nat, natB: Nat), Less (natA: Nat, natC: Nat),
 Less (natB: Nat, natD: Nat)]
```

Den Vergleich, ob `natC` und `natD` identisch sind, müssen Sie innerhalb der Ausführungsmethode des Operators selbst ausführen!

7.4 Operatoren an sich

Nachdem Sie einen aus der abstrakten Klasse `Operator` abgeleiteten Operator mit Hilfe von

```
Operator.setPrecond(Predicate[] prec)
Operator.setPostcond(Postcondition[] postc)
```

erfolgreich mit Vor– und Nachbedingungen[1] versehen haben, kommt nun der schwierigere Teil: die Methode `eval()`. Diese Methode ist nämlich abstrakt und muß von Ihnen überschrieben werden, bevor jemals eine Instanz eines abgeleiteten Operators erzeugt werden kann.

Was `eval` machen soll, ist einfach zu beschreiben: es nimmt sich die Instanzen aus den Prädikatargumenten der Vorbedingung und verarbeitet sie so lange, bis eine Nachbedingung dabei herauskommt, für die eventuell neue Instanzen erzeugt werden müssen. Bevor Sie `eval` wieder verlassen, müssen Sie mit `setPostcondFullfilled (int i)` eine der Nachbedingungen als erfüllt kennzeichnen, wobei die erste Nachbedingung den Index 0 hat.

Wie Sie nun erreichen, daß `eval()` etwas Vernünftiges aus den Vorbedingungen macht, ist ganz alleine Ihr Problem! Hier hängt es von Ihrem programmiertechnischen Können und Überblick ab, wie schnell und wie stabil und ob überhaupt jemals etwas Sinnvolles aus Ihrem Operator herauskommt! Wir wünschen Ihnen dabei auf jeden Fall viel Spaß und Vergnügen! :-)

Bei der Bereitstellung von Konstruktoren müssen Sie beachten, daß Sie immer auch einen Konstruktor mit leerer Parametermenge implementieren (z. B. `MyOperator ()` in der Klasse `MyOperator`. Dieser wird für das Einlesen der Operatorendatenbank (siehe II 8.2) gebraucht.

7.5 Sourcecode mit tausend Worten: ein Beispiel

Um Ihnen einmal zu zeigen, wie ein einfacher Operator aussehen kann, der eine Suchmaschine anfragt, geben wir hier ein kleines Beispiel. Der folgende Operator `NathanSearchOperator` hat die Aufgabe, die Suchmaschine *Nathan*[1] zu befragen. Dazu verlangt der Operator als Vorbedingung ein erfülltes Prädikat

```
searchstring (x : searchstring)
```

wobei `searchstring` ein in unserer T–Box vorkommendes Konzept ist. Der Operator soll weiter zwei Nachbedingungen haben, von denen die erste nur aus einer leeren Add– und Deleteliste besteht, für den Fall, daß der Operator scheitert. Die zweite Nachbedingung hat eine leere Deleteliste und fügt über die Addliste ein Prädikat der Form

[1] Die Klasse `Postcondition` enthält eine jeweils als `Predicate[]` realisierte Add– und Deleteliste

[1] www.nathan.de

```
urlmenge (y : urlmenge)
```

hinzu.

Der Rumpf des Operators ist recht einfach zu realisieren, da in aller Regel nur der Konstruktor und die Methode `eval` implementiert werden müssen:

```
package operators;
  import exceptions.*;
  import informationexchange.LogService;
  import knowledge.*;
  import dataprovider.net.*;
  import dataprovider.search.*;
  import dataprovider.enumerationobjects.*;
  import java.util.LinkedList;

public class NathanSearchOperator
  extends Operator
{
  public NathanSearchOperator () {
    // ...
  }

  public void eval ()
    throws OperatorFailedException {
    // ...
  }
}
```

Die richtige Implementierung des Konstruktors garantiert immerhin schon, daß die Verbindung nach Außen klappt. Alle Planer könnten mit einem solchen Operator arbeiten, da sie ja nur interessiert, was vorne reingeht und was hinten wieder herauskommt. Deswegen ist das Schreiben des Konstruktors, sind Sie sich erst einmal über die gewünschten Vor– und Nachbedingungen im Klaren, eine eher einfache und mechanische Arbeit.

```
public NathanSearchOperator () {
```

Kümmern wir uns zuerst um die Vorbedingung. Davon brauchen wir genau eine:

```
  precond = new Predicate[1];
```

In der Beschreibung des Nathan–Operators haben wir uns ein einstelliges Vorbedingungsprädikat vom Konzept `searchstring` überlegt, daß wir jetzt direkt in der Vorbedingungsliste (in der Implementierung von BotIshelly als Array realisiert) erzeugen:

```
  precond[0] = new Predicate(1,"searchstring");
```

Jetzt müssen wir nur noch ein Prädikatargument mit einem Namen `x` erzeugen und an die einzige Stelle unseres einstelligen Prädikats einfügen (Array–Index 0):

```
  PredArgument x = new PredArgument("x");
  precond[0].setPredArg(x, 0);
```

Das *concept* des Prädikatarguments haben wir nicht zugewiesen, da dieses von der Operatorendatenbank aus dem Namen des einstelligen Prädikats automatisch geschlossen und eingesetzt wird. Genau darum können Sie für Namen einstelliger Prädikate ja auch nur gültige Konzeptnamen benutzen!

Nachdem wir jetzt die Vorbedingung haben, erzeugen wir zuerst eine Nachbedingungslist mit Platz für zwei Nachbedingungen

```
    postconds = new Postcondition[2];
```

und dann zuerst eine Nachbedingung ohne (also mit leeren) Add– und Deletelisten

```
    postconds[0] = new Postcondition();
```

und dann die zweite Nachbedingung mit dem Prädikat `urlmenge` in der Addliste

```
    postconds[1] = new Postcondition();
    Predicate pred = new Predicate(1, "urlmenge");
    PredArgument y = new PredArgument("y");
    pred.setPredArg(y, 0);
    postconds[1].addAddListPred(pred);
  }
```

Das war es auch schon für den Konstruktor. Eigentlich ganz einfach ...

Wesentlich spannender wird es jetzt bei der Methode `eval` die, wir erinnern uns, die ganze Arbeit machen muß. Denn ansonsten kann der Planer einen noch so hübschen Plan aufstellen, der Planausführer kann mit Operatoren, die nichts oder andere Sachen machen, nichts Sinnvolles produzieren.

```
  public void eval ()
    throws OperatorFailedException {
```

Da wir ja nur mit konkreten Instanzen arbeiten können, müssen wir uns erst einmal die (wir haben nur eine) Instanz der Vorbedingung holen. Der String, den wir suchen wollen, steht im Namen der ersten *Instance* im ersten *Predicate* der Vorbedingung, auf den wir mit folgender Kodezeile „direkt“ zugreifen können:

```
    String searchquery = precond[0].getPredArg(0).getInstance().name;
```

Um überhaupt Suchen zu können, müssen wir uns erst einen auf Nathan passenden Suchservice besorgen und uns natürlich auf eventuelle Ergebnisse vorbereiten:

```
    NathanSearchService  service = new NathanSearchService();
    SearchNetResult results;
```

Jetzt sind wir auch schon bereit und können unsere Suchanfrage endlich starten. Danach müssen wir darauf gefaßt sein, diverse Exceptions, die bei der Benutzung des Internets auftauchen können, abzufangen. Das sollte Ihnen aus Java aber bekannt sein, weswegen wie hier nicht weiter darauf eingehen. Anzumerken ist allerdings, daß, falls die Suchmaschine keine Ergebnisse liefert, die leere, erste Nachbedingung (mit dem Array–Index 0) als erfüllt markiert wird:

```
    try {
      results = service.doQuery( searchquery );
    } catch (SearchServiceNoConnectException e1) {
      throw new OperatorFailedException
                              ("Could not connect to SearchEngine");
    } catch (SearchServiceNoResultsException e2) {
      try {
        setPostcondFullfilled(0);
      } catch (exceptions.OperatorNoSuchPostconditionException e3) {
        LogService.logFatal
             (this,"Exception "+e3+": non-existing postcondition!");
      };
      return;   /* "dreckiger" Sprung aus dieser Methode */
    };
```

Das „dreckige“ `return` ist nötig, damit die Methode `eval` im Falle eines Scheiterns sofort verlassen wird.

Nehmen wir einmal an, daß unser Operator nicht gescheitert ist! Die Addliste hat nur ein Prädikat, das wir uns erst einmal „aus den Tiefen“ holen. Zusätzlich erzeugen wir eine neue Java–Instanz der Klasse `Instance`, mit genau dem Konzept des Prädikatarguments in der Addliste:

```
    Predicate postPred =
         (Predicate) postconds[1].getAddList().listIterator().next();
    Instance postPredInstance =
           new Instance(results,postPred.getPredArg(0).getConcept());
```

Jetzt bleibt uns eigentlich nur noch, die neue Instanz auch in das Prädikatargument einzutragen und die zweite Nachbedingung als die erfüllte zu markieren. Den Array–Index 1 haben wir natürlich erzeugt (siehe den Konstruktor weiter oben), müssen aber trotzdem die Exception abfangen, die geworfen würde, wenn der Index nicht existieren sollte:

```
    postPred.getPredArg(0).setInstance(postPredInstance);

    try {
      setPostcondFullfilled(1);
    } catch (exceptions.OperatorNoSuchPostconditionException e4) {
        LogService.logFatal
             (this,"Exception "+e4+": non-existing postcondition!");
    };
  }
```

Das war's! Allerdings müssen wir Sie darauf hinweisen, daß der obige Operator ein sehr einfacher Vertreter seiner Gattung ist. Trotzdem hoffen wir, daß Sie nun eine ungefähre Vorstellung von dem haben, was Sie bei der Programmierung von Operatoren erwartet!

7.6 Spezielle Operatoren

Classifier–Operatoren

Die Klasse `ClassifierOperator` leitet sich direkt aus `Operator` ab und stellt eine Klasse von Operatoren dar, die wohl mit am häufigsten zur Anwendung kommen wird: Operatoren, die eine Eingabe daraufhin untersuchen, ob sie unter ein bestimmtes Konzept fallen. Ein derartiger Operator muß zu jedem Konzept geschrieben werden, zu dem hin Weltobjekte klassifiziert werden sollen; mit Ausnahme von `anything` und `nothing` in der Regel also zu allen Konzepten.

Glücklicherweise weiß jedes T–Box–Konzept, durch welchen Klassifizierer es eine Instanz auf Zugehörigkeit testen kann. Darum ist es möglich, einen allgemeinen `ClassifierOperator` vorzugeben, der nur noch im Konstrukturaufruf

```
ClassifierOperator(Concept postcondConcept,
                   Concept precondConcept)
```

mit den Konzepten des (im Erfolgsfalle) Ergebnisses und der Eingabe gefüttert werden muß.

Der Operator, den Sie erhalten, hat als Vorbedingung ein Prädikat des Konzeptes `precondConcept` und zwei Nachbedingung: ein Prädikat des Konzeptes `postcondConcept` in der Addliste bzw. eine leere Nachbedingung, falls die Eingabe nicht unter das `postcondConcept` klassifiziert werden konnte.

Die `eval()`–Methode ist für alle Classifier–Operatoren gleich und muß von Ihnen nicht neu geschrieben werden. Eigentlich müssen Sie nicht einmal die Classifier–Operatoren selbst schreiben, da Ihnen diese Arbeit von der Operatorendatenbank abgenommen wird (vgl. II 8.3).

Compound–Operatoren

Manchmal kann es sinnvoll sein, Folgen von Operatoren zusammenzufassen. Das könnte z. B. eine immer wieder auftretende Folge bestimmter Operatoren sein. Der Klasse `CompoundOperator` stehen durch Implementierung des `List`–Interfaces Methoden zur Verfügung, um auf den Compound–Operator wie auf eine Liste zuzugreifen. Dadurch ist es möglich, Operatoren der Reihenfolge, in der sie ausgeführt werden sollen, in den Compound–Operator zu speichern. Diese Art von Operatoren soll auch vom Operatorenlerner erzeugt werden.

Bei der Erstellung eines Compound–Operators ist zu beachten, daß ihm an Informationen nur die Dinge zur Verfügung stehen, die er durch die Vorbedingung erhält und solche Informationen, die er selber erzeugt. Die Methode `eval()` arbeitet die enthaltenen Operatoren der Reihe nach ab (muß von Ihnen also nicht mehr geschrieben werden), ohne Vor– oder Nachbedingungen zu belegen. Diese Belegung liegt ganz allein in der Verantwortung des Bauers des Compound–Operators. Also in Ihrer oder in der des Operatorenlerners. Dazu müssen Sie im Konstruktor des Compound–Operators eine Instanz, die z. B. in einem Teiloperator erzeugt wird, in einem anderen als Vorbedingung auftaucht erzeugen und an beiden Stellen in die Prädikatargumente der Vor– und Nachbedingung einhängen. Dazu kann es nötig sein, daß Ihre Teiloperatoren vor dem Erzeugen einer neuen Instanz überprüfen, ob die Instanz vielleicht schon erzeugt ist! Durch das Neuerzeugen einer Instanz würde nämlich der Instanzenfluß unterbrochen und der Compound–Operator immer scheitern!

Die Teiloperatoren werden in serieller Reihenfolge ausgeführt. Es ist nicht möglich, Verzweigungen auszuführen. Das bedeutet aber auch, daß die Anzahl der Nachbedingungen des zuletzt ausgeführten Operators gleich der Anzahl der Nachbedingungen des gesamten Compound–Operators sein muß! Denn es gibt für den Compound–Operator keine Möglichkeit, noch mal abschließend das Ergebnis des letzten Teiloperators zusammenzufassen o. ä. Der Index der im letzten Teiloperator gültigen Nachbedingung ist gleichzeitig auch der Index der gültigen Nachbedingung des gesamten Operators.

Der Compound–Operator scheitert spätestens dann, wenn eine seiner Komponenten scheitert.

Kapitel 8

Operatorendatenbank

Wollten wir alle Herren sein, wer sollte da die großen Säcke tragen?

Aus Dänemark

8.1 Operatorendatenbank im Überblick

Die `OperatorDB` ist eine aus `dataanalysis.databases.GenericDatabase` abgeleitete Datenbank, die die von Ihnen, dem Shellbenutzer, geschriebenen Operatoren enthält. Der Agent kennt nur die Operatoren, die in der `OperatorDB` enthalten sind. Darum muß die Datenbank vor dem ersten Lauf eines Agenten erst einmal mit zu den Anforderungen des Agenten passenden Operatoren gefüllt werden. Die einzelnen Operatoren werden durch eine ID unterschieden, die innerhalb der Datenbank eindeutig ist. Wenn Sie ein und den selben Operator aus Versehen zweimal in die Datenbank speichern, so haben sie den Operator nachher auch doppelt; unter verschiedenen IDs, da diese von der DB fortlaufend vergeben werden. Die Operatoren, die in der Datenbank stehen, sind als Muster zu verstehen, von denen an den Agenten nur jeweils eine Kopie auf Anforderung weitergereicht wird.

8.2 Die Datenbank als Datenbank

Die Operatorendatenbank besitzt zwei Konstruktoren:

```
OperatorDB(T_Box tBox, Concept c, boolean onlySpecialized)
OperatorDB(String file, T_Box tBox, Concept c, boolean onlySpecialized)
```

Die drei Parameter `tBox`, `c` und `specialized` werden im Abschnitt II 8.3 erläutert und dienen der automatischen Generierung der Classifier–Operatoren. Die T–Box dient zusätzlich dazu, die Operatoren aus einer Textdatei (siehe folgenden Absatz) erzeugen zu können.

Wird dem Konstruktor zusätzlich noch ein Dateiname als `String` übergeben, so versucht die Datenbank, ihre Operatoren aus der Datei zu laden. Bei der Datei handelt es sich um eine Textdatei, die zeilenweise wechselnd die Klassennamen und das Gewicht (siehe nächsten Absatz) der Operatoren enthalten. Um z. B. aus dem Paket `operators` die Operatorklasse `PrintlnOperator` mit einem Gewicht von 3.0 zu generieren, so müssen in der entsprechenden Datei die folgenden beiden Zeilen stehen:

```
operators.PrintlnOperator
3.0
```

Das Generieren aus der Datei funktioniert natürlich nur, wenn die Systemvariable `CLASSPATH` das Verzeichnis enthält, in dem sich das Paket `operators` befindet.

Das Gewicht eines Operators gibt an, wie „erfolgsversprechend“ die Anwendung dieses Operators ist und soll dem Planer die Entscheidung zwischen Operatoren erleichtern. Eine Lernkomponente kann aufgrund von Erfahrungen, die mit einzelnen Operatoren gemacht wurden, dieses Gewicht verändern, wobei größere Werte einer höheren Erfolgswahrscheinlichkeit entsprechen. Um das Gewicht eines Operators nachträglich zu ändern, sollte immer die Methode `updateWeightMap (double newWeight, Operator op)` in der `OperatorDB` benutzt werden. Dadurch wird sichergestellt, daß Änderungen am Gewicht eines Operators sich auch auf die Datenbank auswirken, so daß ein später angeforderter Operator derselben ID auch das neue Gewicht hat.

Die aus der Datei geladenen Operatoren werden über den leeren Konstruktur erzeugt, weshalb alle aus `Operator` abgeleiteten Operatoren diesen implementieren müssen. Dabei werden die IDs an die Operatoren nach Reihenfolge des Einlesens aufsteigend vergeben. Die Nummer eines Operators könnte sich also bei einem Neueinlesen ändern, so daß bei der Benutzung der Lernerklassen darauf zu achten ist, mit welcher Version der Operatorendatenbank das entsprechende Planarchiv erzeugt wurde. Eine Änderung der IDs ist allerdings nur zu erwarten, wenn Sie die Datenbank wirklich neu aufbauen und die Operatoren in anderer Reihenfolge angeben. Denn beim Löschen eines Operators aus der DB wird die Stelle einfach freigehalten, so daß keine Verschiebung der IDs eintritt; genausowenig tritt beim Einfügen eines neuen Operators eine Verschiebung ein, da der Operator am Ende der DB eingefügt wird.

Nachdem man die Datenbank gefüllt hat, sei es über eine Datei oder durch Benutzen von `addOperator (Operator newOp)`, wird die Hauptaufgabe der DB darin liegen, auf Anfrage Operatoren zu liefern. Die beiden wichtigsten Methoden dazu sind:

```
public Operator getOperator (long id)
public Operator getNewOperator (long id)
```

Von diesen möchte man in der Regel die zweite aufrufen, da sie eine Kopie des in der Operatorendatenbank unter der Nummer ID gespeicherten zurückliefert, nicht das Original. Der Unterschied ist wichtig, da ein Operator durchaus häufiger als nur einmal angewendet werden kann. Würde auf dem Original gearbeitet, so machten sich Änderungen an der einen Stelle eines Plans gleich an allen Stellen bemerkbar, an denen eine Referenz auf dieses Original steht. Da die Operatoren an anderen Stellen normalerweise auch in anderem Kontext stehen, ist dieser Effekt nicht erwünscht.

Zur Erleichterung stehen speziell für den Planer noch Methoden zur Verfügung, auf die bei der Dokumentation der Klasse `Planner` (siehe II 10.4) noch hingewiesen wird.

8.3 Classifier–Operatoren leichtgemacht

Wie schon angedeutet, die Classifier–Operatoren, die für die einzelnen Konzepte der T–Box gebraucht werden, müssen Sie nicht selber schreiben: diese Arbeit wird Ihnen von der Operatorendatenbank abgenommen. Dazu dienen die drei Parameter `T_Box tBox`, `Concept c` und `boolean onlySpecialized` in den beiden Konstruktoren von `OperatorDB`.

In der übergebenen T–Box finden sich alle Konzepte und Rollen, die wiederum die entsprechenden Klassifizierer kennen, die für sie zuständig sind. Dadurch wird die automatische Generierung der Classifier–Operatoren erst möglich.

Klassifizierer bekommen Instanzen eines Konzeptes als Eingabe und stellen fest, ob es sich hierbei um Instanzen eines Unterkonzeptes handelt. Es wird also immer von einem Oberkonzept zu einem

Unterkonzept hin klassifiziert. Prinzipiell ist zu jeder Kombination aus Ober– und Unterkonzept ein Klassifizierer denkbar. Eine `Ente` kann unter das Konzept `Wasservogel` fallen, das wiederum Unterkonzept zu `Vogel` ist. Statt zwei Klassifizierer `Vogel`⟶`Ente`, `Wasservogel`⟶`Ente` zu bauen, kann man sich auf den allgemeineren ersten Fall beschränken, da damit unter anderem auch Wasservögel als Enten klassifiziert werden können. Gibt es ein gemeinsames Oberkonzept aller Konzepte, zu denen hin klassifiziert werden soll, so reicht es also, wenn A–Box–Instanzen dieses Oberkonzeptes Vorbedingung der Klassifikatoroperatoren sind. Da `anything` alle Konzepte subsumiert, gibt es ein solches Oberkonzept immer. Natürlich muß zu jedem Unterkonzept dieses Oberkonzeptes ein Klassifikator existieren! Daher ist es sinnvoll, das Oberkonzept so speziell wie möglich zu wählen. Da in der Internetdomäne nur URLs als Weltobjekte neu erzeugt werden können, reicht die Wahl des Konzepts `URL` als Obermenge aus. Dieses eine Oberkonzepte wird der Operatorendatenbank im Konstruktor übergeben.

Würde als Oberkonzept `Vogel` übergeben, so erzeugt die Datenbank die Klassifikationsoperatoren zu jedem Unterkonzept von `Vogel`; also auch `Vogel`⟶`Wasservogel` und `Vogel`⟶`Ente`. In manchen Domänen kann es ausreichend sein, nur zu den Blättern des Subsumptionsgraphen hin zu klassifizieren. Wenn Sie also wissen, daß Sie immer A–Box–Instanzen zu den Klassen `Ente`, `Gans`, `Schwan`, usw. haben werden, niemals aber eine A–Box–Instanz zu `Wasservogel`, dann brauchen Sie auch keinen Klassifikationsoperator, der von `Vogel` nach `Wasservogel` klassifiziert (und damit auch keinen Klassifikator). Wenn Sie im Konstruktor der Datenbank `onlySpecialized` mit dem Wert `true` übergeben, so werden nur Klassifikationsoperatoren erzeugt, die vom gemeinsamen Oberkonzept hin zu den spezialisiertesten Unterkonzepten — den Blättern im T–Box–Graphen — gehen. Die nicht erzeugten Klassifikationsoperatoren kennt nun auch der Planer nicht, was den Planungsvorgang beschleunigen kann.

Zu Klassifikatoren, siehe Kapitel II 13, zum Planer Kapitel II 10.

Kapitel 9

Planinformation

Es genügt nicht, zum Fluß zu kommen mit dem Wunsche, Fische zu fangen. Man muß auch das Netz mitbringen.

Aus China

9.1 Planinformation im Überblick

Das Planinformationsobjekt dient zum einen dem Daten– und Informationsaustausch zwischen Systemkontrolle, Planern, Planausführung und Planarchiv. Zum anderen werden durch dieses Objekt die Planungsziele erzeugt und die Ergebnisse der Suche gefiltert.

Die Mächtigkeit dieses Objekts können Sie gut an der Breite des Kontruktors erkennen:

```
PlanInformation (long planID, long searchID,
                 A_Box localABox, T_Box tBox,
                 OperatorDB operatorDB)
```

Zusätzlich zu diesen Angaben wird vom Planer im Planinformationsobjekt der auszuführende Plan abgelegt, der nach erfolgreicher Ausführung und vor dem Abspeichern im Planarchiv in einen Lernbaum umgewandelt wird. Ergänzt werden diese „größeren“ Objekte noch durch einige boolesche Flags, die Auskunft über das Ergebnis der Planausführung geben und/oder bestimmen, wie mit dem Planinfo weiter verfahren werden soll.

Erzeugt wird ein `PlanInformation`–Objekt erstmalig in der Klasse `systemcontrol.Flowcontrol`, die auch die eindeutigen IDs für den planerstellenden Planer (`planID`) und die Suche (`searchID`) generiert.

9.2 Zielextraktion

Die Eingabedaten, die der Agentenbenutzer später an Ihren Agenten schickt, müssen für einen Planer erst in eine sinnvolle Form gebracht werden. Dazu ist es notwendig, die beiden Variablen `targetPredicates` und `startPredicates`, die als `Predicate[]` definiert sind, zu belegen. Dazu werden innerhalb des Konstruktor von `PlanInformation` zwei Methoden aufgerufen:

```
generateTargetPredicates();
generateStartPredicates();
```

Die erste Methode versucht, aus den über `tBox.getTargetConcepts()` und `tBox.getTarget-Roles()` beschafften Informationen über die Benutzereingabe die Prädikate zu erzeugen, die der Planer mindestens mit einem Plan erreichen muß. Leider ist eine automatische Generierung der von der Suchdomäne und den Benutzereingaben abhängigen Zielprädikate zum jetzigen Zeitpunkt nicht möglich, so daß Sie diese Methode überschreiben müssen, damit der von Ihnen zusammengebaute Agent auch weiß, wonach er suchen soll!

In der zweiten Methode werden die Instanzen, die sich in der A–Box schon befinden und somit das Wissen beschreiben, was der Planer voraussetzen darf, in Prädikate umgewandelt. Bitte beachten Sie an dieser Stelle, daß Planer und Planausführer dazu ausgelegt sind, auf Instanzen der Klasse `Predicate` zu arbeiten, die A–Box aber nur auf Instanzen der Klasse `Instance`. Die Umwandlung ist also unbedingt nötig! Diese Methode muß von Ihnen aber nicht überschrieben werden.

9.3 Ergebnisfilterung

Nachdem die Planausführung erfolgreich beendet wurde, sollten die Ergebnisse, die dann hoffentlich vorhanden sind, dem Benutzer auch in irgendeiner Form zur Verfügung gestellt werden. Dazu besorgt sich ***FlowControl*** die Ergebnisse aus dem Planinformationsobjekt über die Methode `getResults()`. Diese Methode ruft intern `verify()` auf, wobei die Implementierung letzterer bei Ihnen liegt!

`verify` belegt die Variable `goalInstances`, ein Array von `Result`, das nachher von `getResults` zurückgegeben wird. Die Aufgabe von `verify` ist es nun, aus der A–Box die Instanzen herauszufiltern, von denen Sie glauben, daß sie der Benutzer gesucht haben könnte. Eine einfache Heuristik wäre, immer die ersten zehn gefundenen Instanzen zurückzuliefern. Sie können die Ergebnisse auch noch einmal klassifizieren und sie dann sortiert nach Relevanz (was auch immer Sie dann als Relevanz definieren) zurückliefern. Auf jeden Fall ist es Ihre Aufgabe, diese Methode für Ihren Agenten entsprechend zu überschreiben!

9.4 Der Planbaum

Der Planbaum stellt eine rekursive Struktur zur Verfügung, in der Operatoren in den Knoten stehen. Jeder Knoten hat so viele Kanten wie Nachbedingungen, wobei bei der Planausführung jeweils der Operator als nächstes ausgeführt wird, der an der Kante der erfüllten Nachbedingung des vorher ausgeführten Operators hängt.

Jeder Planer muß in der Lage sein, einen Planbaum zu erzeugen, auf dem ein Planausführer dann arbeiten kann. Zum erzeugen stehen als wichtigste Methoden der Konstruktor

```
public PlanTree (Operator operator)
```

sowie

```
public PlanTree setNext (PlanTree next, int i)
```

zur Verfügung. Durch den Parameter `i` von `setNext` wird als Index der Nachbedingung des aufrufenden Planbaumknotens interpretiert und der übergebene Planbaum `next` an die entsprechende Stelle eingehängt.

Jede Kante einer Nachbedingung verfügt im Planbaum über zwei boolesche Werte, von denen der eine vom Planer, der andere von der Planausführung gesetzt wird. Ersterer, `goalReached`, kennzeichnet, daß bei Erreichen dieser Nachbedingung im Plan genügend Instanzen bei der Ausführung gesammelt wurden, um das oder die Ziele zu erreichen. Für den Planausführer ist das

das Zeichen, da daß die Suche erfolgreich abgeschlossen wurde. Zweitere Variabe, `goalPath`, wird vom Planausführer gesetzt, wenn die Nachbedingung auf einem Pfad liegt, der zum Ziel geführt hat. In einem Planbaum können durchaus mehrer Pfade als zum Ziel führend markiert sein, da der Planausführer durch Backtracking mehrmals auf unterschiedlichen Wegen zum Ziel kommen kann. Zur Markierung gibt es die Methode `markGoalPath()`, die den Pfad von der erfüllten Nachbedingung bis abwärts zur Wurzel markiert und gleichzeitig bei den Instanzen, die in den Knoten über die PredArgs in den Operatoren hängen, das Flag `successfull` auf `true` gesetzt.

Der Planbaum ist nicht dazu gedacht, in das Planarchiv gespeichert zu werden. Darum muß er vor dem Abspeichern des Planinformationsobjektes mit Hilfe von `PlanInformation.createLearnTree()` in einen Lernbaum umgewandelt werden.

9.5 Der Lernbaum

Der Lernbaum ist eine Variante des Planbaums, die sich von diesem hauptsächlich dadurch unterscheidet, daß sie keine Instanzen mehr enthält. In den Knoten des Lernbaums hängen sogenannte Lernoperatoren, abgespeckte „echte“ Operatoren. Der Lernbaum wurde eingeführt, um nur die Informationen zu speichern, die die Lernerklassen wirklich brauchen.

Da der Lernbaum über die Möglichkeit verfügt, sich als XFig–Grafik auszugeben, kann er von Ihnen sehr gut dazu benutzt werden, Kontrollausgaben der vom Planer erzeugten oder vom Planausführer ausgeführten Pläne zu machen. Der Aufruf von `exportFig ("Kontrolle")` erzeugt so z. B. die Files Kontrolle.fig und Kontrolle.txt, wobei die Nummern in der grafischen Ausgabe den Knotennummern in der textuellen Ausgabe entsprechen. In der grafischen Ausgabe entspricht ein grün gefüllter Kreis einer Nachbedingung, die zum Ziel führte. Alle anderen sind rot. Ist ein Nachbedingungskreis von einem weiteren Kreis umgeben, so ist für diese Nachbedingung das Flag goalReached gesetzt.

Kapitel 10

Planer

> Pläne machen ist mehrmalen eine üppige, prahlerische Geistesbeschäftigung, dadurch man sich ein Ansehen von schöpferischem Genie gibt, in dem man fordert, was man selbst nicht leisten, tadelt, was man doch nicht besser machen kann, und vorschlägt, wovon man selbst nicht weiß, wo es zu finden ist.
>
> *Immanuel Kant*
> Prolegomena 4, 10

10.1 Planer im Überblick

Wenn Sie ein Ziel vor Augen haben, dann können Sie mehrer Arten versuchen, es zu erreichen. Eine Variante wäre, blindlings loszusuchen und zu hoffen, daß Sie schon irgendwann über das eigentlich Gesuchte stolpern werden. Eine andere Möglichkeit ist, sich vorher Gedanken darüber zu machen, wie das Ziel am Besten zu erreichen ist und dem Plan, den Sie dabei erstellen, dann zu folgen. Oft wird es so sein, daß das Gelingen des Plans von allerlei Bedingungen abhängt. Wenn Sie auf eine Art und Weise nicht zum Ziel gelangen, so kommen Sie eventuell auf andere Weise zum Ziel. Manchmal ist es dann notwendig, daß Sie Ihren Plan umschreiben, da Ereignisse eingetreten sind, die Sie nicht vorhergesehen haben. Natürlich kann es auch sein, daß Sie feststellen, daß das von Ihnen gesteckte Ziel gar nicht zu erreichen ist. Stellen Sie das schon bei der Planung fest, so brauchen Sie gar nicht erst loszulaufen.

Einen solchen Plan zu erstellen, das ist die Aufgabe, die eine von der abstrakten Klasse `Planner` abgeleitete Klasse erledigen soll. Der fertige Plan, also die Anweisung, was in welchem Falle zu machen ist, wird dann an den Planausführer gegeben, der den Plan in die Tat umzusetzten versucht. Der Plan, den der Planer erzeugt, muß als `PlanTree` (siehe II 9.4) vorliegen, damit der Ausführer etwas damit anfangen kann.

10.2 Die abstrakte Klasse `Planner`

Aus der abstrakten Klasse `Planner` im Paket `planner` muß der von Ihnen benutzte Planer abgeleitet sein. Als einzige — wenn auch mächtige — Informationsquelle, dient Ihrem Planer dabei das Planinformationsobjekt, was im Konstruktor des Planers übergeben wird. Hier findet der Planer die A– und T–Boxen, die für die Suche relevant sind, hier findet sich die Operatorendatenbank, mit deren Hilfe Operatoren für den Planbaum ausgewählt werden können.

Um zu sicherzustellen, daß eine Systemkontrolle mehrere Planer gleichzeitig beschäftigen kann, ist `Planner` von `Thread` abgeleitet. Daraus folgt aber auch, daß Sie entscheiden müssen, wie oft und an welchen Stellen in Ihrem Planer Sie den Wert der booleschen Flags `destroyThread` und `suspendend` abfragen und darauf so reagieren, daß Ihrem Planer keine Informationen verloren gehen können. Die Zuordnung der Belegung der Flags zu den Aktionen, die ausgeführt werden wollen, sehen so aus:

`destroyThread == true`	$\longrightarrow$	destroy
`suspended == false`	$\longrightarrow$	resume
`suspended == true`	$\longrightarrow$	suspend

Nachdem Ihr Planer von der Systemkontrolle erzeugt wurde, wird die Methode `eval()` aufgerufen, die als Rückgabewert ein `PlanInformation` enthält, daß dann (hoffentlich) einen ausführbaren Plan enthält. Aus diesem Grund müssen Sie in einem eigenen Planer diese Methode überschreiben und mit Sinn, Verstand und wahrscheinlich auch Intuition füllen! Standardmäßig gibt diese Methode nämlich `null` zurück, was Ihnen auf dem Weg zum Ziel nicht viel helfen wird ;-)

Zwei Beispiele für eine Implementierung von `eval` finden Sie in den Klassen `PlannerTypeA` und `PlannerTypeC` im Paket `planner.plannertyes` (s. a. II 10.5 und II 10.6), anhand derer Sie auch die Benutzung des Eventhandlings sehen können.

10.3 Keine Verbindung? Weiterreichen von Instanzen

Ein Planer, der Operatoren zu einem Plan zusammenbaut, möchte in aller Regel, daß der Planausführer dort später Daten fließen läßt. Diesen Datenfluß muß aber bereits der Planer sicherstellen, da der Ausführer naturgemäß keine Ahnung davon hat, was sich ein Planer beim Aufstellen des Plans so alles gedacht hat. Im Normalfall wird der Ausführer den Ersteller des Plans nicht einmal kennen.

Wenn Ihr Planer einen Operator anwendet, dann soll das Ergebnis meistens weiterverwendet werden; vor allem dann, wenn Ihr Operator neue Weltobjekte erzeugt. Darum muß der Planer schon konkrete Java–Instanzen der Klasse `Instance` erzeugen, auch wenn er sie noch nicht mit Inhalt füllt oder füllen kann. Damit diese Instanz von Operator A zu einem Operator G weitergereicht werden kann, muß der Planer diese Instanz schon in die entsprechenden Prädikatargumente einhängen. Dadurch ist sichergestellt, daß der Planausführer, wenn er die Instanz in Operator A mit Inhalt füllt, beim Erreichen von Operator B schon eine Referenz auf eine Instanz vorfindet, nämlich genau die aus Operator A. Dann wird der Ausführer auch nicht mehr versuchen, dieses Instanz mit einem anderen Inhalt zu füllen.

Wenn Sie also einen eigenen Planer schreiben wollen, dann müssen Sie darauf achten, daß der Instanzenfluß gewährleistet ist!

10.4 Hilfestellung durch die Operatordatenbank

Eine Hilfestellung erhalten Sie allerdings durch die Operatorendatenbank, die Ihnen einige Funktionen zur Verfügung stell, mit denen Sie eine Auswahl aus den möglichen Operatoren treffen können. Da Sie Operatoren mit Ihren Vor– und Nachbedingungen zu einem Planbaum zusammenstecken müssen, können Sie sich mit den beiden Methoden

```
getAllOperators4PrecondPred (Predicate p)
getAllOperators4PostcondPred (Predicat p)
```

einen `ListIterator` auf alle Operatoren zurückgeben lassen, die das Prädikat `p` in der Vor– bzw. Nachbedingung haben.

Es kann sinnvoll sein, zuerst die Operatoren mit der besten Gewichtung anzuwenden. Auf diese können Sie mit der Methode

```
getBestOperators ()
```

zugreifen, die einen `ListIterator` über die Operatoren, absteigend sortiert nach Gewichtung, zurückliefert. Die Methode `getBestOperators` liefert übrigens schon Klone der Operatoren aus der Datenbank, Sie müssen also nicht anhand der ID mit `getNewOperator (ID)` noch eine Kopie anfordern. Die Verwendung der Operatoren mit höchster („bester") Gewichtung stellt auf einfache Weise sicher, daß Operatoren, die durch die Lernkomponenten als „erfolgreich" bewertet wurden, häufiger angewendet werden, da sie weiter vorne in der Liste stehen. So kann Ihr einmal geschriebener Planer auf die Ergebnisse der Lerner reagieren, ohne daß Sie den Planerkode ändern müssen.

10.5 Planer Typ A

Die Klasse `PlannerTypeA` implementiert einen Planer, der jeweils einen „Plan" erstellt, der aus einem Operator besteht, diesen zur Ausführung bringt und aufgrund des Ergebnisses den nächsten Schritt plant. Dieser Planer ähnelt reaktiven Planern, unterscheidet sich von diesen aber deutlich durch die mangelnde Fähigkeit, Sensoren zur Orientierung einsetzen zu können. Klassische Sensoren („Welches Signal zeigt die Ampel?") haben im Internet scheinbar keine Entsprechung.

Der Planer Typ A verfolgt eine sehr einfache Strategie, um das Ziel zu erreichen: die Operatoren werden in der Reihenfolge ihrer Gewichtung ausprobiert. Nach einer gewissen Anzahl von Schritten wird versucht, einen speziellen Operator anzuwenden, der als Vorbedingung die Zielprädikate hat. Kann dieser Operator von der Planausführung erfolgreich instantiiert werden, so war der Plan erfolgreich und die Suche kann beendet werden. Die Suche ist auch beendet, allerdings abgebrochen, wenn der Planer feststellt, daß er alle möglichen Operatoren hintereinander ausgeführt hat, *ohne* daß sich sein Wissen über die Welt verändert hat. Da das Internet für die Dauer der Suche als statisch angesehen wird, würde auch keine weitere Anwendung der Operatoren ein neues Ergebnis liefern.

Bemerkenswert ist der Planer vom Typ A, weil er in seiner Strategie einfach, in seiner Handhabung aber recht anspruchsvoll ist. Im Gegensatz zu „echten" Planern, die einen Plan erstellen und damit fertig sind, ist der Typ A–Planer auf die Ergebisse angewiesen, die der Planausführer liefert. Da der Aufruf des Planausführers nur über die Systemkontrolle geschehen kann, benutzt der Planer die Events

```
planReadyUnfinished
planExecuteFinished
```

um die gewünschte Weiterverarbeitung zu kontrollieren. Dabei löst das Event `planReadyUnfinished` als Reaktion aus, daß der Planer suspended und die Planausführung mit dem Planinformationsobjekt gestartet wird. Der Planer hat vorher im Planinformationsobjekt das Flag `backToSender` gesetzt, was später den Planausführer veranlassen wird, das Info–Objekt über die Systemkontrolle wieder an den ursprünglichen Planer zurückzuschicken.

Das zweite Event, `planExecuteFinished`, besagt der Systemkontrolle, daß der Planer Typ A seine Aufgabe erfüllt hat. Die Systemkontrolle beendet den Planer und kann daraufhin entscheiden, ob der fertige Plan ein Teilplan aus einem größeren Kontext war und einen anderen Planer mit der Weiterbearbeitung beschäftigen (vgl. dazu auch den Abschnitt über das Eventhandling in der Systemkontrolle).

10.6 Planer Typ C

Der Planer Typ C ist eine Variante von GraphPlan [Blum und Furst, 1997], der als derzeit schnellster Planalgorithmus für STRIPS–beschränkte klassische Planungsdomänen gilt. Der Hauptunterschied zum klassischen GraphPlan besteht in der Variante, die `PlannerTypeC` benutzt (PG 343–GP), besteht in der Hinzunahme dynamisch wachsender Objektwelten und dem Zulassen disjunktiver Nachbedingungen.

Um Operatoren mit mehreren Nachbedingungen anwenden zu können, werden diese intern zu Nachbedingung–vielen Operatoren aufgesplittet, wobei die i–te Nachbedingung auf den i–ten Operator abzielt. Durch diese Modifikation greift der ursprüngliche GP–Algorithmus.

Nach Erstellen des Plans wird das Event `planReadyFinished` ausgelöst, was die Weiterreichtung des Planinformationobjektes an eine Planausführung veranlaßt.

Kapitel 11

Planausführung

> Es ist im Leben wie im Schachspiel: Wir entwerfen einen Plan; dieser bleibt jedoch bedingt durch das, was im Schachspiel dem Gegner, im Leben dem Schicksal zu tun belieben wird. Die Modifikationen, welche hierdurch unser Plan erleidet, sind meistens so groß, daß er in der Ausführung kaum noch an einigen Grundzügen zu erkennen ist.
>
> *Arthur Schopenhauer*
> Aphorismen zur Lebensweisheit V, 48

11.1 Planausführung im Überblick

Aus eigener Anschauung wissen Sie wahrscheinlich, daß derjenige, der einen Plan aufstellt, diesen nicht unbedingt auch persönlich ausführen muß. Bei BotIshelly ist es so, daß der Planer keinerlei Möglichkeit hat, selber den entworfenen Plan auszuführen. Hier gibt es die Klasse `PlanExecution`, die ein Planinformationsobjekt entgegennimmt und den dort enthaltenen Planbaum abzuarbeiten versucht.

Grob vereinfacht füllt die Planausführung dazu die Vorbedingung des ersten Operators mit Instanzen aus der A–Box und führt ihn dann aus. Als nächstes wird überprüft, welche Nachbedingung gültig geworden ist und entsprechend zum nächsten Operator verzweigt, bei dem das Spiel wieder von vorne losgeht. Im günstigsten Fall erreicht der Planer eine Nachbedingung, für die `goalReached` gesetzt ist. Dann ist das Ziel erreicht und die Präsentation der Ergebnisse kann von der Systemkontrolle veranlaßt werden.

11.2 Die Klasse `PlanExecution`

Die Klasse `PlanExecution` ist zur Abwechslung mal nicht abstrakt, und Sie müssen auch keine `eval`– oder ähnliche Methode selber schreiben! Der Planausführer von BotIshelly arbeitet mit den Planbäumen ganz zufriedenstellend zusammen ;-)

`PlanExecution` ist aus `Thread` abgeleitet und wird von der Systemkontrolle mittels `run()` gestartet. Vorher muß der Planausführer natürlich über den Konstruktor

```
PlanExecution (PlanInformation planInfo, int maxInstances)
```

erzeugt werden. Alle Informationen, die den Plan und das Wissen betreffen, findet der Planausführer im Planinformationsobjekt. Die zusätzlich übergebene Beschränkung `maxInstances` gibt an, wieviele Instanzen höchstens bei einem Backtracking in Betracht gezogen werden. Dadurch wird verhindert, daß ein Operator, der als Vorbedingung `anything` hat, mit einer unter Umständen sehr großen, kompletten A–Box belegt wird und dadurch Ihren Rechner (oder anderer Leute Rechner) lahmlegt.

Wenn Sie den Planausführer nur ein wenig verbessern wollen, so können Sie am ehesten an den Methoden

```
LinkedList selectInstances (Concept c)
LinkedList selectConcreteRoles (Role r)
```

ansetzen. Diese beiden Methoden besorgen sich aus der A–Box die `maxInstances` ersten Konzepte bzw. konkreten Rollen, die zurückgeliefert werden. Hier können Sie versuchen, aus den Instanzen eine Auswahl zu treffen, die z. B. die mit den höchsten Konfidenzwerten bzgl. des zu belegenden Konzeptes oder der Rolle.

Erreicht der Planausführer eine Nachbedingung, deren `goalReached` gesetzt, ist, so könnte er aufhören und den Pfad im Planbaum als zum Ziel führend markieren. Allerdings wird vom Benutzer in aller Regel mehr als nur ein Ergebnis erwartet. Darum wird an der Stelle ein Backtracking eingeleitet und der Plan bis zu der Stelle zurückgerollt, an der noch eine ausgewählte Instanz nicht benutzt worden war. Dabei müssen Sie natürlich, wollen Sie den Planausführer umschreiben, darauf achten, daß die A–Box auch nach dem Backtracking noch konsistent ist. Insbesondere müssen Sie auf die Instanzen achten, die durch die Add– oder Deletelisten eines Operators hinzugekommen oder gelöscht worden sind!

Die in BotIshelly gewählte Implementierung des Planausführers stellt sicher, daß nur für Instanzen an den Stellen, an denen sie zum ersten Mal in einem Pfad im Planbaum auftauchen, über mögliche Belegungen iteriert wird. Alle weiteren Vorkommen einer Instanz sind aufgrund der vom Planer zu gewährleistenden Gleichheit der Java–Referenzen in den entsprechenden Prädikatargumenten für den Planer schon belegt und werden nicht neu iteriert. Wenn Sie eine eigene Planausführung schreiben wollen, dann müssen Sie hierauf achten! (Vgl. dazu Abschnitt II 10.3)

11.3 Hilfestellung durch die Instanzen und die A–Box

Um das im vorhergehenden Abschnitt angesprochene Problem, hinzugekommene Instanzen löschen zu müssen, gelöschte Instanzen wieder hinzufügen zu müssen, kommen Ihnen die Klassen `Instance` und `A_Box` ein Stückchen entgegen. In `Instance` gibt es das boolesche Flag `delete`, das von der Planausführung gesetzt oder wieder gelöscht werden darf. Eine Instanz, die innerhalb einer Deleteliste auftaucht, müssen Sie also nicht wirklich löschen, sondern können sie nur als gelöscht markieren. Wenn Sie dann ein Backtracking ausführen, setzten sie das Flag auf `false` und Sie haben wieder den alten Zustand erreicht. Sind Instanzen neu hinzugekommen, so markieren Sie sie im Backtrackingschritt als deleted.

Die A–Box reagiert auf das Flag `deleted` in so weit, als Sie von der A–Box keine Konzepte oder konkreten Rollen zurückgeliefert bekommen, in denen eine der Instanzen als gelöscht markiert ist. Nach dem Backtracking verhält sich die A–Box also so, als wären die Instanzen wirklich gelöscht worden. Wie ist das nun mit Instanzen, die über eine Addliste hinzugekommen sind und deren `delete` im Laufe eines Backtrackings auf `true` gesetzt worden ist? Das ist auch kein Problem, da es hier zwei Möglichkeiten gibt:

1. Die Instanz lag nicht auf einem Pfad, der zum Ziel geführt hat.
2. Die Instanz lag auf einem Pfad, der zum Ziel geführt hat.

In erstem Fall ist es nicht schade um die Instanz, da sie uns nicht weitergebracht hat. Im zweiten Fall jedoch ist gleichzeitig für die Nachbedingung, in der die Instanz entstanden ist, das Flag `goalPath` im Planbaum gesetzt. Dadurch wird die A–Box beim Abspeichern daran gehindert, diese Instanz wirklich zu löschen.

11.4 Mengen von Instanzen

Manchmal möchte man Operatoren schreiben, die eine Menge von Instanzen zurückliefern. So ist es z. B. sinnvoll, wenn ein Operator, der eine Suchmaschine befragt, alle Suchergebnisse zurückliefert. Die Prädikatargumente sind in BotIshelly in der Lage, Mengen von Instanzen zu speichern.

Da der Planer die Pläne anhand abstrakter Instanzen erzeugt, ist die Mengenwertigkeit der Prädikatargumente für ihn nicht interessant. Darum ändert sich auch nichs in der Formulierung der Vor– und Nachbedingungen eines Operators. Anders liegt der Fall beim Planausführer. Dieser muß sehr wohl darauf achten, ob ein Operator in einem Prädikatargument eine oder mehrere Instanzen abgelegt hat!

Neue Instanzen für die A–Box — und damit auch Mengen von Instanzen — können nur innerhalb von Operatoren erzeugt werden. Wenn der Operator nur eine Instanz für ein Prädikatargument erzeugt, dann wird mit der Methode ***PredArgument***`.setInstance` diese Instanz gesetzt. Mengen von Instanzen können durch wiederholtes Aufrufen von ***PredArgument***`.addInstance` hinzugefügt werden.

Kapitel 12

Datenbeschaffung

Wer kein Messer hat, kann kein Brot schneiden
aus Spanien

Die Klassen im Paket `dataprovider` stellen für einen Agenten die Schnittstellen zum Netz dar. Grundlegende Funktionen sind fertig imlpementiert. So ist z. B. für Agenten, die auf HTML–Dokumenten arbeiten, die via HTTP geladen werden, keine weitere Programmierarbeit nötig.[1] Es existieren auch Klassen, deren Instanzen die Abfrage von Suchmaschinen im WWW ermöglichen.

Ferner gibt es die Möglichkeit, Dateien aus dem lokalen Dateisystem zu laden, um z. B. die Klassifikatoren mit lokalen Daten trainieren zu können, ohne dafür einen lokalen HTTP–Server einrichten zu müssen.

12.1 Wichtige Klassen im Überblick

In den folgenden Abschnitten werden verschiedene, wichtige Klassen der Datenbeschaffung vorgestellt, und es wird beschrieben, wie diese zu benutzen sind.

`NetService` Die Net–Services stellen die Möglichkeit bereit, den Inhalt einer URL herunterzuladen. Zur Zeit werden die Protokolle HTTP und FTP, sowie der Zugriff auf lokale Dateien unterstützt.

`NetEntity` Diese Klasse stellt die Superklasse, den Grundtyp aller Objekte dar, die etwas aus dem Netz repräsentieren, wie z. B. Links, HTML–Seiten u. ä..

`NetResult` In Objekten dieses Typs werden die von einem *`NetService`* heruntergeladenen Dateien bzw. Dokumente abgelegt. Über verschiedene Methoden kann auf Teilaspekte eines Dokumentes zugegriffen werden, wie z. B. auf Titel und Links einer HTML–Seite.

`SearchService` Mit den Search–Services können Suchmaschinen auf einfache Art und Weise abgefragt werden.

Wenn die vorhandenen Klassen bzw. deren Funktionalität nicht ausreichen, so gibt es in einem gesonderten Abschnitt Tips, wie und wo Erweiterungen möglich sind, und was dabei zu beachten ist.

[1] bezogen auf die Datenbeschaffung natürlich

12.2 `dataprovider.net.NetService` — Realisierung von Transferprotokollen

Die Net–Services stellen die Funktionalität bereit, Daten aus dem Netz durch Angabe ihrer URL herunterzuladen. In der aktuellen Version von BotIshelly werden die drei Protokolle HTTP, FTP und File[1] unterstützt, die mit Hilfe der drei Klassen `HTTPNetService`, `FTPNetService` und `FileNetService`[2] benutzt werden können. Alle Protokolle sind als Subklassen der abstrakten Klasse `NetService` realisiert. Dies sollte auch für weitere, neu zu programmierende Protokollservices so gehandhabt werden.

Um zu einer URL, deren Protokoll bekannt ist, den passenden Service zu erhalten, kann direkt ein Objekt der entsprechenden Klasse instanziieren werden, z. B.:

```
HTTPNetService  service = new HTTPNetService();
```

Wenn das Protokoll nicht bekannt ist, bzw. noch nicht feststeht, kann die statische Methode `NetService.getNetService` benutzt werden. Diese Klassenmethode liefert den zur übergebenen URL passenden Service zurück oder wirft eine Exception, falls deren Protokoll nicht unterstützt wird. Im Programmcode könnte dies wie folgt aussehen:

```
NetService  service;
String url;

try {
  // Funktioniert
  url = "http://www.nathan.de/";
  service = NetService.getNetService( url );

  // Funktioniert auch
  url = "file:/a/b/c.html";
  service = NetService.getNetService( url );

  // Funktioniert nicht
  url = "news:fbi-news";
  service = NetService.getNetService( url );
}
catch ( NetServiceNoSuchServiceException e ) {
  System.out.println( "Unbekanntes Protokoll in URL " + url );
}
```

Nachdem ein entsprechender Service instanziiert worden ist, kann dessen Methode `getNetResult` aufgerufen werden, um den Inhalt einer übergebenen URL herunterzuladen. Vorher kann via `setTimeout` ein Timeout (in Sekunden) gesetzen werden, der festlegt, nach welcher Zeit eine Verbindung, die nicht antwortet, abgebrochen wird.[3] Sollte die gewünschte Information passwortgeschützt sein, müssen über die Methoden `setUser` und `setPassword` der Benutzername, respektive das Passwort gesetzt werden.

Das folgende Beispiel zeigt, wie eine Datei, die auf einem FTP–Server liegt, heruntergeladen werden kann. In diesem Beispiel würde allerdings eine Exception geworfen werden, da gzip'te Postscriptdateien von BotIshelly nicht unterstützt werden.

[1] File ist natürlich kein echtes Protokoll. Der Einfachheit halber wird hier jedoch keine (weitere) Unterscheidung vorgenommen.

[2] Der widersprüchliche Name kommt daher, daß lokale Dateien in der Modellierung noch nicht vorgesehen waren.

[3] Bei einem *`FileNetService`* wird der Timeout ignoriert.

```
FTPNetService service = new FTPNetService();
NetResult result;

try {
  service.setUser ("anonymous");
  service.setPassword ("user@somewhere.com");
  service.setTimeout (30);
  result = service.getNetResult("ftp://ftp.de/pub/memo47.ps.gz");
}
catch (Exception e) {
  // hat wohl etwas nicht funktioniert...
}
```

12.3 dataprovider.net.NetEntity — Der kleinste Nenner im Datenaustausch

Die abstrakte Klasse `NetEntity` stellt die Superklasse für Objekte dar, die etwas aus dem Netz modellieren, und die zum Informations- und Datenaustausch zwischen den unterschiedlichen Funktionseinheiten eines Agenten vorgesehen sind. Beispiele sind Links, HTML–Seiten oder auch Bild- und Binärdateien. Unter anderem kommunizieren die Datenbeschaffung, die Klassifikatoren und die A–Box(-en) mit Hilfe dieser Klasse.

Eine ***NetEntity*** verkörpert also ein Objekt, das dem Netz "entsprungen" ist, und dessen "Ursprugsort", nämlich die URL, die Methode `getURL` zurückliefert. Damit eine ***NetEntity*** z. B. als Ergebnis einer Suche auch ausgegeben bzw. angezeigt werden kann, existiert die abstrakte Methode `getShortDescription`, die eine für diesen Zweck sinnvolle Beschreibung liefern sollte.

Bei einer zielgerichteten Suche ist es wichtig, herausfinden zu können, ob eine ***NetEntity*** überhaupt etwas mit dem Suchziel zu tun hat, bzw. aus welchen Gründen auch immer für wichtig oder unwichtig erachtet werden sollte. Es muß also die Möglichkeit der Klassifikation bestehen. Aus diesem Grund sind die Methoden `addClassification`, `getClassification` und `getAllClassificators` implementiert. Außerdem ist die abstrakte Methode `getText` vorhanden, die eine textuelle Beschreibung der ***NetEntity*** liefern sollte, die den Klassifikatoren eine möglichst gute Präzision ihrer Entscheidungen ermöglicht. Nachdem die Klassifikatoren ihre Ergebnisse über `addClassification` eingetragen haben, kann auf diese schließlich im weiteren Programmablauf via `getClassification` und `getAllClassificators` zugegriffen werden.

12.4 dataprovider.net.NetResult — Grundtyp aller Repräsentationen von Dateien bzw. Dokumenten

Die abstrakte Klasse `NetResult` ist die Superklasse aller möglichen, von BotIshellyunterstützten Dokument- bzw. Dateitypen. Sie ist folgerichtig eine Subklasse von `NetEntity`. Das Ergebnis der Methode `getNetResult` eines Net–Services ist — falls keine Exception geworfen wurde — eine Instanz einer Subklasse von `NetResult`, z. B. ein ***HTMLTextNetResult*** bei HTML–Dokumenten. Zur Verdeutlichung die Hierarchie der Klassen[1]:

Sollen neue Dokumenttypen unterstützt werden, so muß eine entsprechende, eventuell neu zu erstellende Subklasse von `NetResult` benutzt werden. In der Methode `getNetResult` eines jeden

[1]Ein dünner Rahmen zeigt an, daß es sich um eine abstrakte Klasse handel.

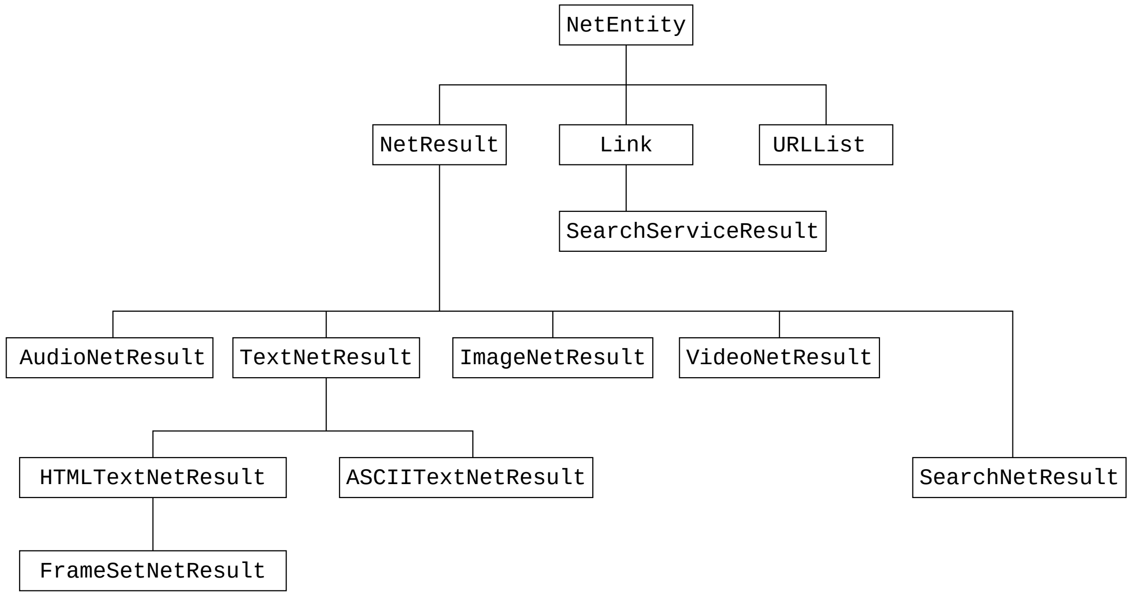


Abbildung 12.1: Hierarchie der Subklassen von `NetEntity`

Net–Services, der diesen Typ unterstützen soll, muß dies vorgesehen werden, d. h. der Programmcode dieser Methode muß erweitert werden. Wie dies genau zu geschehen hat, wird noch erleutert werden.

Die Klasse `NetResult` definiert verschiedene Methoden, um auf Einzelaspekte der dargestellten Datei zugreifen zu können:

`getDownloadDate` Liefert das Datum (als *java.util.Date*), an dem die Übertragung abgeschlossen wurde.

`getSubmittedDate` Liefert das Datum der letzten Änderung der Datei, soweit das benutzte Protokoll eine solche Information vorsieht, sonst das gleiche Datum wie `getDownloadDate`.

`getRaw` Mit dieser Methode kann der "rohe", ungeänderten Inhalt der heruntergeladenen Datei als *String* angefordert werden. Sie wird selten gebraucht werden, da z. B. zum Ermitteln der Links einer HTML–Seite bereits eine Methode existiert.

`getShortDescription` Diese Methode liefert eine Kurzbeschreibung des Inhaltes eines *NetResult*. Z. B. bei HTML–Dateien die ersten 300 Zeichen des Textes der Seite.

`getText` Von `NetEntity` geerbte Methode, die die Information liefert, nach der klassifiziert wird.

`getURL` Ebenfalls von `NetEntity` geerbte Methode. Liefert die URL, mit der der *NetResult* geladen wurde.

12.4.1 Die Klasse `HTMLTextNetResult`

Die Klasse `HTMLTextNetResult` soll noch besondere Erwähnung finden, da sie zum einen besondere Funktionen anbietet und zum anderen HTML wohl das meistbenutzte Dokumentenformat im WWW ist. So kann die Methode `getLinks` dazu verwendet werden, alle Links einer HTML–Seite zu ermitteln:

```
HTMLLinkEnumeration enumeration;
HTTPNetService service = new HTTPNetService;
HTMLTextNetResult result;
Link link;

try {
  // Datei runterladen
  result = (HTMLTextNetResult) service.getNetResult("http://www.linux.com");

  // Enumeration erzeugen
  enumeration = result.getLinks();

  // Die einzelnen Ergebnisse ausgeben
  while (enumeration.hasMoreElements()) {
    link = (Link) enumeration.nextElement();
    System.out.println ("URL=" + link.getURL() );
    System.out.println ("Text=" + link.getLinkText() );
  }

  // Zum Schluß noch den Titel der Seite anzeigen
  System.out.println ("Titel der Seite: " + result.getTitle());
}
```

Die Methode `getLinks` liefert eine *`Enumeration`* für alle Links einer HTML–Seite. Die einzelnen Objekte in dieser Aufzählung sind vom Typ `Link`. Auf die URL und den Link–Text eines *`Link`* kann über die Methoden `getURL` und `getLinkText` zugegriffen werden.

Bei HTML–Seiten kann es auch sinnvoll sein, den Titel einer Seite zu ermitteln. Dieses ist über die Methode `getTitle` möglich.

Im WWW ist es inzwischen üblich, in HTML–Seiten sogenannte "Frames" zu benutzen, was zur simultanen Darstellung von mehreren HTML–Dateien in einem WWW–Browser–Fenster führt. Die Angabe einer URL kann bei einer Suche per Hand also zur Darstellung weiterer Seiten führen, die zur Informationsgewinnung dienen können, ohne daß deren URL je sichtbar wird. Diesem Sachverhalt trägt die Klasse `FrameSetNetResult` Rechnung, welche eine Subklasse von `HTMLTextNetResult` ist. Bei Instanzen dieser Klasse, wurden alle HTML–Dateien, die ein WWW–Browser anzeigen würde, geladen. Die Methoden `getText` und `getLinks` verhalten sich entsprechend.

12.5 dataprovider.search.SearchService — Abfragen von Suchmaschinen leicht gemacht

Über die Search–Services wird eine Schnittstelle zu Suchmaschinen bereitgestellt. Alle verfügbaren Suchmaschinenschnittstellen sind dabei Subklassen der abstrakten Klasse `SearchService`. In der aktuellen Version von BotIshelly sind die Suchmaschinen AltaVista und Nathan abfragbar (www.altavista.com, www.nathan.de).

Eine exemplarische Abfrage der Suchmaschine AltaVista zeigt der folgende Beispielcode:

```
SearchResultEnumeration enumeration;
AltavistaSearchService service = new AltavistaSearchService();
SearchServiceResult result;
SearchNetResult results;
```

```
try {
  results = service.doQuery ("java compiler");
  enumeration = results.getResults();

  while (enumeration.hasMoreElements()) {
    result = (SearchServiceResult) enumeration.nextElement();

    System.out.println("URL=" + result.getURL());
    System.out.println("Desc=" + result.getShortDescription());
    System.out.println("Title=" + result.getText());
  }
}
catch (Exception e) {
  // wohl ein Fehler aufgetreten...
}
```

Zuerst muß eine Instanz der entsprechenden `SearchService`–Subklasse erzeugt werden, in diesem Fall also von `AltavistaSearchService`. Danach wird deren Methode `doQuery` aufgerufen, der die einzelnen Suchworte durch Freizeichen getrennt übergeben werden. Die Search–Services führen dann auf diesen Suchworten eine UND–verknüpfte Anfrage durch.

Das Ergebnis der Methode `doQuery` ist ein *`SearchNetResult`*. Um die einzelnen Suchergebnisse (i. d. R. zehn Stück) zu erhalten, muß zunächst über die Methode `getResults` eine *`SearchResultEnumeration`* erzeugt werden. Wie bei einer *`java.util.Enumeration`* vorgesehen, kann nun mit den Methoden `hasMoreElements` und `nextElement` auf die einzelnen Objekte der Aufzählung, in diesem Fall also die einzelnen Suchergebnisse, zugegriffen werden. `nextElement` liefert hier Instanzen von `SearchServiceResult`.

Bei einem *`SearchServiceResult`* kann schließlich über die Methoden `getURL`, `getText` und `getShortDescription` auf die URL, den Linktext und die Kurzbeschreibung des Suchergebnisses zugegriffen werden.

12.6 Modifikationen & Erweiterungen

12.6.1 Die Klasse `HTMLConvert`

Mit den Klassenmethoden `convertEntities` und `convertToEntities` kommen Sie in der Regel nicht in Kontakt, sie seien hier aber erwähnt, da sie bei Erweiterungen der Shell vielleicht nützlich sein können.

Entities[1] werden in Markup–Sprachen verwendet, um Zeichen zu kodieren. Insbesondere in HTML finden diese Verwendung: Da es z. B. nicht erlaubt ist, das Zeichen ´`&`´ zu verwenden, muß stattdessen ´`&`´ benutzt werden.

Sollten in einem *`String`* solche Entities vorkommen, können diese mit der Klassenmethode `convertEntities` aufgelöst werden. Sollte der umgekehrte Weg gewünscht werden, um z. B. einen *`String`* mit Sonderzeichen korrekt in einem Browser auszugeben, realisiert die Klassenmethode `convertToEntities` eben dieses.

12.6.2 `NetService`: Unterstützung eines neuen Protokolls

Sollten neue Protokollservices hinzugefügt werden, so muß sich dies auch in der Klassenmethode `NetService.getNetService` widerspiegeln, damit deren Funktionalität erhalten bleibt, d. h. der

[1] Hier nicht in Zusammenhang mit der Klasse `NetEntity`

Programmcode dieser Methode muß erweitert werden. Die Stelle, an der dies zu geschehen hat ist deutlich durch Kommentare gekennzeichnet.

Um ein neues Protokoll zu unterstützen, muß zunächst die entsprechende Klasse als von `NetService` erbende Klasse realisiert werden. Danach muß in dieser neuen Klasse die Methode

```
public NetResult getNetResult (String url)
  throws NetServiceException, MalformedURLException
```

implementiert werden, der eine URL als *`String`* übergeben wird und die ein *`NetResult`* zurückliefert.

Als Subklasse von `NetService` erbt die neue Klasse auch die `String`–Variablen *`user`* und *`password`* (für passwortgeschützte Dokumente; siehe die Methoden `setUser` und `setPassword`) sowie die `int`–Variable *`timeout`*, die den Timeout der Datenverbindung in Sekunden angibt. Beim Implementieren der Methoden sollten diese drei Variablen natürlich beachtet und ihrer Bestimmung entsprechend genutzt werden.

Zuerst sollte geprüft werden, ob die URL überhaupt korrekt ist. Im Fehlerfall sollte eine `MalformedURLException` ausgelöst werden, was z. B. mit der Zeile

```
new java.net.URL (url);
```

möglich ist, da der `URL`–Konstruktor genau diese Exception werfen kann/darf.[2]

Danach wird die Datei heruntergeladen und in einem *`String`* abgespeichert. Je nach Protokoll muß dann noch entschieden werden, welchen Typ der heruntergeladene Inhalt hat. Möglichkeiten wären der Content–Type bei HTTP–Verbindungen, die Dateiendung bei FTP–Verbindungen oder die Betrachtung der ersten 16 Bytes (siehe `file`–Kommando bei UNIX-Systemen). Außerdem müssen zwei Zeitpunkte (als *`java.util.Date`*) bekannt sein: das Datum der letzten Änderungen (für `submittedDate`) und das Datum, an dem die Übertragung abgeschlossen wurde (für `downloadDate`). Mit diesen Daten wird dann die Klassenmethode `composeNetResult` aufgerufen. Als erster Parameter muß dieser Methode ein String übergeben werden, der den Typ der geladenen Datei beschreibt. Bisher werden einige MIME–Typen und einige Dateinamenendungen erkannt. Welche Typen erkannt werden, wird von der Abbildung bestimmt, die durch die statische *`TreeMap`* `typeMap` in der Klasse `NetService` induziert wird. Das Ergbenis dieser Methode kann schließlich als Ergebnis der Methode `getNetResult` zurückgeben werden. Die Exceptions die `composeNetResult` werfen kann, können — und sollten — dabei einfach weiterpropagiert werden.

Als gutes Beispiel für die Implementierung der Methode `getNetResult` empfiehlt sich das Studium der Datei `HTTPNetService.java`, in der z. B. das Last–Modified–Attribut als `submittedDate` genutzt wird.

12.6.3 `NetResult`: Hinzufügen eines neuen Dokumenttyps

Um einen neuen Dokumenttyp hinzuzufügen, muß zuerst eine neue Klasse in die `NetEntity`– bzw. `NetResult`–Hierarchie eingesetzt werden. Danach sind die geerbten, abstrakten Methoden (z. B. `getText`) zu implementieren. U. U. kann es auch sinnvoll sein, geerbte Methoden zu überschreiben, oder auch zu überladen.

Damit die vorhandenen Net–Services Ihren neuen Dokumenttyp benutzen, d. h. Instanzen Ihrer Klasse erzeugen, können, müssen in der Klasse `NetService` folgende Erweiterungen getätigt werden:

[2] D. h. keine `try`/`catch`–Klammerung um diesen Aufruf.

1. Die MIME–Typen und die Dateinamenendungen, die dem neuen Dateityp entsprechen, müssen in die `typeMap` eingetragen werden, indem sie auf einen *`Integer`* abgebildet werden, der einen noch nicht benutzten Wert repräsentiert. Dies passiert in einem statischen Codeblock von `NetService`, z. B.:

   ```
   // MIME-type für HTTPNetService
   typeMap.put("text/vnd.wap.wml", new Integer(333) );
   // file-extension für FTPNetService
   typeMap.put("wml",              new Integer(333) );
   ```

2. In der Methode `composeNetResult` muß obiger Integerwert einen "Switch–Case" einleiten, in dessen Anweisungsteil dann eine Instanz Ihrer neuen Klasse erzeugt und zurückgeben werden muß, z. B.:

   ```
   case 333:
     return new WMLNetResult(url, rawContent, download, lastModified);
   ```

12.6.4 `SearchService`: Hinzufügen einer neuen Suchmaschinenschnittstelle

Um eine neue Suchmaschine hinzufügen zu können, muß diese zwei Bedingungen erfüllen:

1. Sie muß über eine GET–Anfrage abfragbar sein.
2. Zu jedem Suchergebnis muß sie eine URL, einen Titel und eine Kurzbeschreibung liefern.

Nehmen wir eine fiktive Suchmaschine, deren Suchergebnisseiten so aussehen:

```
<html>
...

<dl>
<dd><a href="http://www.test.de/">Erste Seite</a>
<dt>Kurbeschreibung zur ersten Seite<br>

<dl>
<dd><a href="http://www.test2.de/">Zweite Seite</a>
<dt>Kurbeschreibung zur zweiten Seite<br>
...

</html>
```

Wie bei allen (bekannteren) Suchmaschinen üblich, sind die Suchergebnisse nach einem Schema aufgebaut (bei Suchmaschinen, für die das nicht gilt, muß ein neuer Parser geschrieben werden, der die abstrakte Klasse `ParserService` implementiert). Bei obigem Beispiel gilt, daß die URL immer zwischen `<dd><a href="` und `"` steht. Für die drei Bestandteile URL, Titel und Kurzbeschreibung müssen Sie nun nach solchen umschließenden Zeichenketten suchen.

Ihre neue Klasse muß von der abstrakte Klasse `SearchService` erben:

```
public class NewSearchService extends SearchService
{
    public SearchNetResult doQuery(String queryString)
```

```
                  throws SearchServiceNoResultsException,
                         SearchServiceNoConnectException
    {
        SimpleParserService  parser = new SimpleParserService();
        SearchServiceResult  result;
        ParserStore          store;
        NetResult            netresult;
        String               url;

        //...
    }
}
```

Danach definieren Sie einen neuen Parser, dem Sie reguläre Ausdrücke für die umschließenden Zeichenketten (Format: RegulärerAusdruckLinks|Variable|Regulärer Ausdruck rechts) übergeben können. In unserem Beispiel:

```
try {
  parser.setParser( new String[]  { "<dd><a href=\"|URL|\"",
                                    ">|TITLE|</a>",
                                    "<dt>|DESC|<br>" } );
}
catch( ParserServiceMalformedParserException e ) {
   // ...
}
```

Als nächstes müssen Sie die URL erzeugen, mit der die Suchmaschine via GET–Request abgefragt wird. Achten Sie dabei vor allem darauf, daß Sie gemäß CGI–Konvention kodieren (ein Leerzeichen wird zu `+`, etc.). Wie Sie das mit Hilfe der `Transformer`–Klasse bewerkstelligen können, lesen Sie einfach im Quellcode der Klasse `AltavistaSearchService` nach.

```
url = "http://www.somewhere.com/cgi-bin/query.cgi?QUERY=" + queryString;
```

Dann können Sie den Inhalt dieser URL herunterladen und darauf den Parser anwenden:

```
/* Erstmal nen Service ermitteln - sollte HTTPNetService sein */
try { netservice = NetService.getNetService( url ); }
catch( Exception e ) { throw new SearchServiceNoConnectException(); }

/* Runterladen - zumindest versuchen */
try { netresult = netservice.getNetResult( url ); }
catch( Exception e ) { throw new SearchServiceNoConnectException(); }

/* Inhalt der HTML-Seite */
try{ content = netresult.getRaw(); }
catch( NetResultReloadFailedException e ) {}

/* Parsen */
try { store = parser.parseString( content ); }
catch( ParserServiceUnknownParserException e ) {}

/* Ein SearchNetResult erzeugen */
result = SearchNetResult( store, url, content,
                          netresult.getDownloadDate(),
                          netresult.getSubmittedDate() ) );
```

Das *`SearchNetResult`* enthält jetzt die Suchergebnisse. Wie bereits beschrieben, kann jetzt mit der Methode `getResults` eine *`Enumeration`* über die einzelnen Suchergebnisse erhalten werden. Im übrigen muß an dieser Stelle noch geprüft werden, ob überhaupt Suchergebnisse gefunden wurden (ein *`SearchNetResult`* kann auch mit null Suchergebnissen erzeugt werden). Falls keine Ergebnisse vorliegen, muß die Methode `doQuery` die Exception `SearchServiceNoResultsException` auslösen.

Kapitel 13

Datenanalyse

> Die Natur hat uns zwar viele Kenntnisse versagt, sie läßt uns über so manches in einer unvermeidlichen Unwissenheit, aber den Irrtum verursacht sie doch nicht. Zu diesem verleitet uns unser eigener Hang, zu urteilen und zu entscheiden auch da, wo wir wegen unserer Begrenztheit zu urteilen und zu entscheiden nicht vermögend sind.
>
> *Immanuel Kant*
> Logik 8

13.1 Aufgabe der Datenanalyse

Die Datenanalyse und damit die Klassifikatoren beschäftigen sich in der ersten Linie mit der Analyse bzw. der Klassifizierung von Texten, die von der Datenbeschaffung übergeben wurden.

Für die Klassifizierung werden Klassifizierer bereitgestellt auf die später noch im einzelnen eingegangen wird. Damit Sie auch noch eigene Ideen einbinden und verwirklichen können, besteht zusätzlich die Möglichkeit eigene oder neu entwickelte Klassifikatoren in das System zu integrieren. Wie dies genau geschieht wird Ihnen noch näher erläutert. Zunächst wird die Funktionalität der einzelnen Komponenten der Datenanalyse beschrieben.

13.2 Arbeitsweise

Die Datenanalyse stellt verschiedene Klassifikatoren zur Verfügung, die sich im wesentlichen mit der Analyse von Texten befassen. Die bereits implementierten Klassifikatoren sind der Bayes–Klassifikator (Klassifizierung von Text mit wahrscheinlichkeitstheoretischen Ansätzen), der k–NN-Klassifikator (Entwickelt in der Statistik, benutzt die Definition eines Abstandsmasses) und der Rocchio–Klassifikator (Vektorraum–Retrieval-Modell, hierbei werden Dokumente als Wortvektoren dargestellt).

Für speziellere Aufgaben gibt es zusätzlich noch den `NetEntityClassifier`, den `LinkPageClassifier` und den `URLClassifier`. Die zuletzt genannten Klassifikatoren zeichnen sich dadurch aus, daß sowohl das Training, wie auch die Klassifikation auf eine eindeutige Aufgabe hin orientiert sind.

Alle Klassifikatoren können in ihrer Arbeitsweise trainiert werden, indem sie bereits bewertete Trainingsmengen (Dokumente) klassifizieren und einordnen. Wichtig ist hierbei, daß sowohl positive wie auch negative Beispiele eingesetzt werden, um den größtmöglichen Lernerfolg zu erzielen. Die Trainingsmengen können von Ihnen zusammengestellt und bewertet werden. Je genauer Sie an diese Aufgabe herangehen, desto besser werden die Ergebnisse.

Während der Klassifizierung wird neben dem Feature–Wörterbuch (Grundlage für den `Documentvector`) auch noch das Negativ–Wörterbuch(Stop–Wort–Liste) und eventuell das Synonym–Wörterbuch eingesetzt.

Die Wörter des Negativ–Wörterbuchs werden bei der Umwandlung eines Textes in einen Vektor nicht berücksichtigt, während die Wörter aus dem Synonym–Wörterbuch dabei helfen können die Suche mit verwandten Wörtern erweitert durchzuführen. Weitere Informationen zum Thema Vektorisierung finden sie auf der Seite 198.

Auch der Vektorisierer, der für die Erstellung eines Vektors aus einem Text verantwortlich ist, steht den Klassifikatoren als wichtiges Hilfsmittel zur Verfügung. Viele Klassifikatoren nutzen die Darstellung von Texten als Vektoren.

Nachdem die Systemkontrolle die `ClassifierDatabase` geladen hat, können die Klassifikatoren eingesetzt werden. Der Planer hat hierbei die Möglichkeit für die Durchführung seines Plans einen konkreten Klassifikator aus der Klassifikatoren–Datenbank (`ClassifierDatabase`) zu laden (Bsp.: `get(Klassifikatorname)`). Nachdem die Auswahl getroffen ist kann die Klassifizierung beginnen.

Soviel zu der allgemeinen Arbeitsweise der Klassifikatoren, in den nun folgenden Kapiteln wird Ihnen nun genauer erklärt werden, wie das Training eines Klassifikators abläuft, die Klassifizierung der einzelnen Dokumente funktioniert und welche Hilfsmittel Ihnen bereits zur Verfügung stehen.

Für die Implementierung eines neuen Klassifikators müssen die ab Seite 202 aufgeführten Punkte berücksichtigt werden.

13.3 Die Konstruktion von Klassifikatoren

Zunächst betrachten wir die Klasse, die allen hier implementierten Klassifikatoren, zugrundeliegt. Die Klasse `Classifier` stellt das Grundgerüst für die Klassifikatoren dar. Jeder `Classifier` hat die Möglichkeit auf den `Vectorizer` oder das `Dictionary` zurückzugreifen. Der `Vectorizer` ist ein wesentliches Hilfsmittel für die Klassifizierer. Der Vektorisierer transformiert die textuelle Darstellung eines Dokuments in einen vektoriellen Wert, den `Documentvector`. Dabei hat der Vektorisierer zwei unterschiedliche Aufgaben, die sich bereits anhand des Methodenaufrufs erkennen lassen.

Befindet sich der Klassifikator im Trainingsmodus, so erfolgt der Aufruf durch `vectorize(newDocument, true)`. Der boolesche Parameter läßt hierbei erkennen, daß sich der Vektorisierer zur Zeit im Trainingsmodus befindet. Was bedeutet nun Trainingsmodus? In der Trainingsphase ist der Klassifikator bemüht so viele Merkmale wie möglich aus den Beispieldatenmengen herauszufiltern. Merkmale bedeutet in diesem Kontext, die unterschiedlichen Wörter aus den Dokumenten zu sammeln und in einem Wörterbuch dem sogenannten Feature–Wörterbuch einzubinden.

Da sich der `Documentvector` an diesem speziellen Wörterbuch orientiert sollte eine Veränderung der Wörterbücher während der eigentlichen Klassifikationsphase unterbunden werden, dies wird durch den Methodenaufruf `vectorize(newDocument, false)` erreicht.

Um die Funktionsweise und den Aufbau des `Documentvectors` zu verdeutlichen, stellen wir Ihnen die grundlegenden Aspekte vor. Wie bereits erwähnt untersteht die Klasse `Documentvector` in erster Linie dem Vektorisierer. In diesem Zusammenhang werden alle wichtigen Methoden zur

Verfügung gestellt, die es erlauben ein Dokument in einen Vektor zu konvertieren und auch Berechnungen zwischen zwei Vektoren durchzuführen. Als Beispiele sind hier eine kleine Auswahl vorhandener Methoden aufgeführt: `setElement`, `addElement`, `dimension`, `dotProduct`, `euklidLength`.

Aus Platzgründen wird der `Documentvector` spärlich repräsentiert, d.h. Positionen die einen Wert von 0.0 beinhalten werden nicht explizit gespeichert. Dies bedeutet auf der anderen Seite, daß alle unbesetzten Stellen des Vektors automatisch den Wert 0 enthalten.

Die Dokumentvektoren sind auch für die Klassifizierer von größter Bedeutung, da fast alle Berechnungen der Abstände oder der Ähnlichkeit über Vektoren ermittel werden. Wenn Sie einen neuen Klassifizierer implementieren möchten, so wäre es sinvoll, wenn sie sich einen Überblick über die Methoden der Klassen `Vectorizer` und `Documentvector` verschaffen. Vielleicht kann hierdurch ein zusätzlicher Programmieraufwand verhindert werden, indem bereits vorhandene Methoden eingesetzt werden.

13.3.1 Aufbau eines Dokumentvektors

Der Dokumentvektor ist das Ergebnis der Vektorisierung eines Textes (String). Bei den hier vorgestellten Klassifikatoren wird der Dokumentvektor wie folgt konstruiert:
Der übergebene String (Dokument) wird in seine Wörter zerlegt. Dabei wird ein `StringTokenizer` eingesetzt, der dafür sorgt, daß aus dem Text, sämtliche Notation herausgefilter wird und nur noch die Wörter selbst zurückbleiben.

Für jedes einzelne Wort wird zunächst überprüft, ob es im Negativ–Wörterbuch (Stopwort–Wort–Liste) vorkommt. Ist dies der Fall, so wird das aktuelle Wort bei der Vektorisierung nicht berücksichtigt. Im anderen Fall, wird das Wort ins Feature–Wörterbuch eingetragen. Anhand der Position, an der das Wort ins Wörterbuch eingetragen wurde, erfolgt der Eintrag in den Dokumentvektor. Kommt ein Wort häufiger im Text vor, so wird der Wert an der Stelle im Dokumentvektor um eins erhöht, ein erneuter Eintrag ins Feature–Wörterbuch erfolgt nicht.

Wenn der Klassifikator sein Training durchführt, durchläuft ein Text nach dem anderen den Vektorisierer. Dabei wächst das Feature–Wörterbuch und somit auch die Dimension des Dokumentvektors kontinuierlich an. An dieser Stelle sei nochmals darauf hingewiesen, daß die Anpassung der Wörterbücher nur im Traingsmodus der Klassifikatoren möglich ist.

Um den Aufbau eines Dokumentvektors zu veranschaulichen folgt ein Beispiel:

Übergebener Text: „Die Klasse Dokumentvektor untersteht in erster Linie dem Vektorisierer. In diesem Zusammenhang werden alle wichtigen Methoden zur Verfügung gestellt, die es erlauben ein Dokument in einen `Documentvector` zu konvertieren.“

Bei der Zerlegung des Textes werden nur die Wörter berücksichtigt, die Notation in dem Dokument (String) wird vernachlässigt. Für die Konstruktion des Dokumentvektors stehen drei verschiedene Wörterbücher zur Verfügung.

- **Feature–Wörterbuch**: Grundlage für den `Documentvector`, da er sich an den Positionen der Wörter in diesem Wörterbuch orientiert.
- **Negativ–Wörterbuch**: Dieses Wörterbuch wird auch häufig Stop–Wort–Liste genannt. Alle Wörter, die in diesem Wörterbuch aufgeführt sind, werden bei der Klassifizierung nicht berücksichtigt. (mögliche Beispiele: und, der, ein usw.)
- **Synonym–Wörterbuch**: Hier besteht die Möglichkeit bei der Klassifizierung auch verwandte Wörter mit einzubeziehen, dabei erhofft man sich eine erhöhte Treffsicherheit bei der Klassifizierung.

Zustand der Wörterbücher vor der Vektorisierung:

Wörterbuch	Inhalt
Feature–Wörterbuch	—
Negativ–Wörterbuch	ein, einen, die, dem, diesem, es, untersteht, gestellt, erlauben, in, zur, werden, erster, alle, wichtigen

Zustand der Wörterbücher nach der Vektorisierung:

Wörterbuch	Inhalt
Feature–Wörterbuch	Klasse, Dokumentvektor, Linie, Vektorisierer, Zusammenhang, Methoden, Verfügung, Dokument, konvertieren
Negativ–Wörterbuch	ein, einen, die, dem, diesem, es, untersteht, gestellt, erlauben, in, zur, werden, erster, alle, wichtigen

Resultierender Dokumentvektor:

Position	Wort	Anzahl
1	Klasse	1
2	Dokumentvektor	2
3	Linie	1
4	Vektorisierer	1
5	Zusammenhang	1
6	Methoden	1
7	Verfügung	1
8	Dokument	1
9	konvertieren	1

(1, 2, 1, 1, 1, 1, 1, 1, 1) ist der Dokumentvektor des übergebenen Dokuments.

Nachdem nun die Klasse `Documentvector` eingehend beschrieben wurde, können wir uns nun dem Teil widmen, der diesen Vektor einsetzt, der Klasse `Vectorizer`. Bei der Vektorisierung können drei verschiedene Wörterbücher eingesetzt werden (Feature–, Negativ– und Synonym–Wörterbuch), deren Grundgerüst aus der Klasse `Dictionary` stammt, und die alle notwendigen Methoden beinhaltet, die beim Erzeugen und Bearbeiten von Wörterbüchern notwendig sind. Die Wörterbücher sind transient definiert um den Speicherplatz effektiver nutzen zu können. Im einzelnen können die Bücher über den gesetzten Namen aus der `DictionaryDatabase` aufgerufen werden.

Um den Trainingsmodus eines Klassifikators von der eigentlichen Klassifikation zu trennen, hilft, wie oben im Text bereits angedeutet, ein boolescher Parameter in der `vectorize`-Methode aus der Klasse `Vectorizer`.

Die Methode `vectorize` erhält als zweiten Parameter neben dem zu klassifizierenden Text auch noch einen booleschen Wert. Befindet sich der Klassifikator im Trainingsmodus so wird `vectorize` mit dem Zusatz `true` aufgerufen. Somit wird die Anpassung der Wörterbücher ermöglicht. Dies bedeutet, daß mit jedem neuen Dokument aus der Trainingsmenge auch eine Veränderung der Wörterbücher (insbesondere des Feature-Wörterbuchs) einhergeht.

Da sich aber die Dimension des Dokumentvektors an der Anzahl der Wörter im Feature-Wörterbuch orientiert, muß eine weitere Anpassung der Wörterbücher während der eigentlichen Klassifizierung unterbunden werden. Dies wird erreicht indem die `vectorize`-Methode nun mit dem zu klassifizierenden Dokument und dem Zusatz `false` aufgerufen wird. In diesem Fall ist es nicht mehr möglich, daß sich die Wörterbücher automatisch anpassen und somit die Dimension des Vektors verfälschen könnten.

Bevor wir zur Konstruktion neuer Klassifikatoren kommen, werden Ihnen die bereits implementierten Klassifikatoren kurz vorgestellt.

13.3.2 Rocchio-Klassifkator

Der Rocchio–Algorithmus wurde als Methode zum Relevance Feedback im Rahmen des Information–Retrieval–Systems SMART entwickelt. Der Algorithmus basiert auf dem Vektorraum–Retrieval–Modell. In diesem Modell werden Dokumente ebenso wie Anfragen als Wortvektoren repräsentiert. Weiterhin wird ein Abstandsmaß definiert, welches die (semantische) Ähnlichkeit dieser Vektoren messen soll. Wird eine Anfrage an das Retrieval–System gestellt, werden die Dokumente aus der Datenbank anhand ihrer so definierten Ähnlichkeit zur Anfrage sortiert. Es wird davon ausgegangen, daß Dokumente um so wahrscheinlicher relevant in Bezug auf die Anfrage sind, je ähnlicher sie zu ihr sind.

Ablauf des Rocchio–Algorithmus:

- Dokumente werden vektorisiert (Vektoren enthalten die TF-Werte der Worte).
- Nachdem alle Dokumente vektorisiert wurden beginnt die Gewichtung über einen zweiten Faktor: Inverse Document Frequency

  $$IDF(w) = log\frac{n}{DF(w)}$$

  (n: Gesamtzahl der betrachteten Dokumente, DF: Anzahl der Dokumente, die das Wort w mindestens einmal enthalten). Um die Anzahl der Dokumente herauszufinden, die das Wort w enthalten, müssen das Feature–Wörterbuch und sämtliche Vektoren durchlaufen und überprüft werden. Anhand des Feature–Wörterbuchs kann ein Array erzeugt werden, daß die IDF–Werte zu den Wörtern abspeichert.
- TFIDF–Gewichtung berechnen: Je häufiger ein Wort in einem Dokument vorkommt, desto wichtiger (TF–Teil) und wenn das Wort in vielen Dokumenten vorkommt desto unwichtiger kann es sein (IDF–Teil).
- Um den d–Vektor zu erhalten muss nun die TF–Repräsentation des Textes mit der IDF–Repräsentation der Wörter aus dem Feature–Wörterbuch elementweise multipliziert werden.
- Um die Ähnlichkeit zwischen Dokumenten herauszufinden wird der Kosinus der zugehörigen Vektoren benutzt.

13.3.3 Bayes-Klassifikator

> Der Bayes–Klassifikator basiert auf einer Modellierung von Text mit wahrscheinlichkeitstheoretischen Methoden. Es wird davon ausgegangen, daß Texte von verschiedenen Wahrscheinlichkeitsverteilungen erzeugt werden. Diese Verteilungen sind sehr komplex und lassen sich direkt nicht handhaben. Deshalb werden vereinfachende Annahmen gemacht, die es ermöglichen, die Verteilungen auf einem Computer darzustellen und aus Trainingsdaten zu schätzen.

Quelle: [Joachims, 1997]

13.3.4 k–NN-Klassifikator (k–Nearest–Neighbor–Verfahren)

> Bei dem k–Nearest–Neighbor-Verfahren handelt es sich um einen Klassifikatorenalgorithmus, der in der Statistik entwickelt wurde. Im Unterschied zu dem naiven Bayes'schen Klassifikator werden keine Verteilungsannahmen gemacht. Vorwissen kann allerdings an anderer Stelle, nämlich bei der Definition des sogenannten Abstandsmaßes eingebracht werden. Dieses Abstandsmaß (wird häufig mit dem Cosinus zweier Dokumentvektoren berechnet) wird dazu benutzt, ähnliche Situationen aus der Vergangenheit zu finden. Es sollte so beschaffen sein, daß Dokumente, die anhand dieses Maßes ähnlich sind, möglichst zur selben Klasse gehören. Die Grundidee des k–Nearest–Neighbor–Verfahrens ist es, ein neues Beispiel so zu klassifizieren wie die ähnlichsten k Beispiele, für die man die Klassifikation kennt.

Quelle: [Joachims, 1997]

Wie groß k im Einzelfall sein soll, kann von Ihnen selbst bestimmt werden. Mit der Methode `setK (int i)` aus der Klasse `KNNClassifier` wird der Wert für k gesetzt. Wird k nicht explizit übergeben, so erhält k automatisch den Wert fünf. Um den aktuellen Wert von k auszulesen kann die Methode `getK()` aufgerufen werden.

Da die Datenanalyse nicht immer komplizierte Klassifizierer benötigt, gibt es noch drei speziellere Klassifikatoren, die für bestimmte Einsatzgebiete benutzt werden können.

Da wäre zum einen der `LinkPageClassifier`. Dieser Klassifikator ist in der Lage, HTML-Seiten anhand der in der Seite vorkommenden Links zu klassifizieren. Sollte die Mehrheit der Links zu denen im Training angegebenen Konzepten gehören, so wird die Seite als positiv eingestuft.

Als nächstes gibt es dann noch den `NetEntityTypeClassifier`, der anhand des Objekt-Typs `NetEntity` klassifiziert. Sollte der Klassenname der übergebenen `NetEntity` oder einer Elternklasse dieser `NetEntity` einem Name entsprechen, mit dem der Klassifizierer trainiert wurde, so wird 1.0 eingetragen, ansonsten 0.0. Wird der Klassifizierer z.B. mit `dataprovider.net.TextNetResult` trainiert, so werden alle davon erbenden Klassen sowie die Klasse selbst als positiv klassifiziert.

Zum Schluß gibt es noch den `URLClassifier`. Hier werden URL´s anhand von Wortvorkommen klassifiziert. Alle Worte die in den Beispielen vorkommen, werden auf Vorkommen in der URL der übergebenen `NetEntity` überprüft. Sollte mindestens eines der Worte vorkommen, trägt der Klassifikator 1.0 ein, ansonsten 0.0. Die Groß–/Kleinschreibung wird nicht beachtet.

Falls Ihnen diese Klassifikatoren nicht ausreichen, so besteht die Möglichkeit weitere Klassifizierer zu implementieren. Was Sie dabei beachten müssen zeigt Ihnen das unten aufgeführte Beispiel mit den aufgeführten Erklärungen.

13.3.5 Implementierung neuer Klassifikatoren

Bei der Implementierung eines neuen Klassifikators muß zunächst darauf geachtet werden, daß es mindestens zwei Methoden gibt. Dies wäre zum einen die Trainingsfunktion (`public int train ()`) und zum anderen die eigentliche Klassifikationsmethode (`public NetEntity classify (NetEntity document)`).

Die Trainingsdaten werden von der Datenbeschaffung bereitgestellt und sind unter `untrainedExamples` (aus der Klasse classdataprovider.net.NetEntity) abgelegt. Diese Beispielmenge besteht aus mehreren Strings, wobei jeder einzelne ein Dokument darstellt der zusätzlich noch eine Bewertung enthält, die das Beispiel als positiv oder negativ einordnet.

Diese vorherige Bewertung der einzelnen Beispiele muß von Ihnen vorgenommen werden, und je gewissenhafter Sie bei Durchführung vorgehen, desto besser sind die Ergebnisse bei der Klassifikation. Um Beispiel hinzuzufügen wird die Methode `addExample` (Übergabe eines Strings oder eines Objekts vom Typ `NetEntity`) aus der Klasse `Classifier` benutzt.

Dies wäre der klassische Weg. Im Laufe der Entwicklung dieses Agenten, wurden zur schnelleren Einbindung von Beispieldaten Hilfsmittel (Tools) entwickelt, die Ihnen die Einbindung von Trainingsmengen und die von Ihnen vorzunehmende Bewertung der einzelnen Bespiele erleichtern sollen. Aus diesem Grund verweise wir auf das Kapitel Tools.

Sind Trainingsbeispiele vorhanden so können diese dafür benutzt werden über den Vektorisierer (Methode: `vectorize (String document, boolean autoAddWord)`), die Wörterbücher für ihre spezielle Aufgabe anzupassen. Um diese Anpassung zu erreichen muß dem Vektorisierer neben dem Text auch noch ein `true` mitgeliefert werden. `true` sorgt dafür, daß das Feature–Wörterbuch angepaßt wird, da gerade das Feature–Wörterbuch für den Aufbau des Dokumentvektors verantwortlich ist.

Die untrainierten Beipiele aus der Menge `untrainedExamples` der Klasse `dataprovider.net-.NetEntity` müssen aus Speicherplatzgründen nach dem durchgeführten Training gelöscht werden (`untrainedExamples.clear()`). Dabei besteht die Möglichkeit als Rückgabewert die Anzahl der trainierten Beispiele auszulesen.

Nachdem sämtliche Dokumente vektorisiert wurden, kann damit begonnen werden, die Texte jeweils als positives oder negatives Beispiel einzuordnen. Um eine Grenze zu setzen, ab der ein Dokument als positiv oder negativ bezeichnet wird, kann die Methode `setThreshold` aus der Klasse `Classifier` benutzt werden. Hierbei muß beachtet werden, daß der Schwellwert zwischen 0.0 und 1.0 liegt und daß die Dokumente mit einer Bewertung über diesem Grenzwert als zugehörig (also positiv) eingestuft werden. (Der Aufbau eines Dokumentvektors wird ab Seite 199 beschrieben).

Für einige Klassifikatoren ist es wichtig, daß die Vektoren der übergebenen Beispiele in TF–IDF–Form vorliegen. Die Berechnung wurde bereits im Ablauf des Rocchio-Algorithmus beschrieben (Seite 201).

An dieser Stelle ist das Training abgeschlossen. Die gewonnenen Daten (Wörterbücher und Dokumenvektoren) können nun bei der folgenden Klassifizierungsmethode eingesetzt werden.

Die Klassifikationsmethode (`public NetEntity classify (NetEntity document)`) bekommt von der Datenbeschaffung ein Dokument übergeben, das klassifiziert werden soll. Werden unzulässige Werte eingetragen oder eine Exception eingeleitet, so kann der Klassifizierer seine Analyse nicht durchführen. Daraus folgt, daß der Klassifikator kein Ergebnis eintragen kann. Alle transienten Variablen müssen mit der Methode `checkVariables()`

überprüft werden. Die benötigten Wörterbücher und Vektorisierer können aus der `ClassifierDatabase` geladen werden. Zunächst wird das übergebene Dokument vektorisiert. Dabei muß darauf geachtet werden, daß die Vektorisierungsfunktion mit dem Zusatz `false` gestartet wird. `false` sorgt dafür, daß eine Anpassung der Wörterbücher nun nicht mehr stattfindet. (Möglicherweise ist auch an dieser Stelle eine Umwandlung des Vektors in eine TF–IDF–Form notwendig. Hierfür können die bereits in der Trainingsmethode benutzten Funktionen eingesetzt werden.)

Nach der Vektorisierung stehen alle Mittel zur Verfügung, die für eine Einordnung des neuen Dokuments benötigt werden. In der Klasse `Dokumentvector` sind Methoden abgelegt, die die Berechnungen zwischen verschiedenen Vektoren erleichtern.

Nachdem die Klassifikation durchgeführt wurde kann das Ergebnis mit dem Befehl `document.addClassification (getElementname(), classifyResult)` gespeichert werden.

13.4 Grobstruktur eines neuen Klassifikators

Hier bekommen Sie einen Überblick, wie die Struktur eines Klassifikators auszusehen hat. Es sind dabei nur die wesentlichen Punkte aufgeführt, die Ihnen helfen sollen einen eigenen, neuen Klassifizierer in das System einzubinden.

```
// Source file: sourcen/dataanalysis/classifiers/newClassifier.java

package dataanalysis.classifiers;

import dataprovider.net.NetEntity;
import dataanalysis.docvectors.*;
import dataanalysis.databases.*;
import dataanalysis.dictionaries.*;
import informationsexchange.LogService.*;

public class newClassifier {

// hier koennen alle lokalen Variablen definiert werden

// Konstruktor
public newClassifier() {
 super();
 ...}
// Konstruktor mit Datenbankuebergabe
public newClassifier(ClassifierDatabase theDB)
 super(theDB);
 ...}

public int train(){
...
// Ueberpruefung, ob die transienten Variablen gesetzt wurden
error = checkVariables();
...
// Nachdem alle Trainingsdokumente vektorisiert wurden, koennen
// die Dokumente geloescht werden
untrainedExamples.clear();

// Da die Woerterbuecher nun angepasst sind, koennen die notwendigen
// Berechnungen nun durchgefuehrt werden
...
return (Anzahl-Dokumente);
}

public NetEntity classify(NetEntity document){
// das Dokument wird anhand der Daten aus dem Trainings eingeordnet
// und klassifiziert
...
// Eintragung des Klassifikationsergebnisses
document.addClassification(getElementname(),classifiesValue);
...
return (document);
}
}//newClassifier
```

13.5 Operationen auf der ClassifierDatabase

Zum Schluß noch ein Beispiel, wie Sie einen Klassifizierer aus der `ClassifierDatabase` laden oder speichern können.

```
public void load(String fname)
  throws ClassNotFoundException, IOException
{
  FileInputStream fs = new FileInputStream(fname);
  ObjectInputStream is =  new ObjectInputStream(fs);
  theDB = (ClassifierDatabase)is.readObject();
  is.close();
};

public void save(String fname)
  throws IOException
{
  FileOutputStream fs = new FileOutputStream(fname);
  ObjectOutputStream os = new ObjectOutputStream(fs);
  os.writeObject(theDB);
  os.close();
}
```

Benutzung im Programm:

```
try {
  load("Daten.dat");
} catch(Exception e) {
  e.printStackTrace();
  System.exit(2);
};
```

Kapitel 14

Instanzenlerner

> Menschen von dem ersten Preise
> lernen kurze Zeit und werden weise.
> Menschen von dem zweiten Range
> werden weise, lernen aber lange.
> Menschen von der dritten Sorte
> bleiben dumm und lernen Worte.
>
> *Konfuzius*

Der Instanzenlerner wird nach der Beendigung einer Suchanfrage von der Systemkontrolle aufgerufen, und hat in erster Linie die Aufgabe den Inhalt der verschiedenen lokalen A–Boxen zu untersuchen und in der globalen A–Box zusammenzuführen. Um die lokalen A–Boxen zu Verfügung zu haben, arbeitet der Instanzenlerner auf dem Planarchiv, das unter anderem die lokalen A–Boxen enthält.
Als erstes ruft die Methode `InstanceLearner.update` die Methode `A_Box.update` einer solcher lokalen A–Box auf. Dieser Aufruf bewirkt, daß die lokale A–Box bereinigt wird. Das heißt, alle Instanzen, die nur *`anything`* oder den unter *`startConcepts`* zusammengefaßten Konzepten zugeordnet werden konnten, werden gelöscht. Genau so wie alle Instanzen, deren *used*–Falg auf false steht und alle Instanzen, deren *delete*–Falg auf true und das *successful*–Flag auf false seht. Im zweiten Schritt durchläuft der Instanzenlerner die lokale und die globale A–Box parallel und fügt neue Instanzen hinzu, bzw. erhöht bei schon vorhandenen Instanzen den `Instance.counter`–Wert um eins und setzt bei `Instance.date` das aktuelle Datum. Zum Schluß löscht der Instanzenlerner alle lang nicht mehr benutzten Instanzen. Dazu berechnet er anhand des *delete–time*-Wertes das Datum, ab wann Instanzen als lang nicht mehr benutzt gelten.
Der Einsatz des Instanzenlerners hat den Effekt, daß die Suchzeit bei ähnlichen Anfragen sehr viel geringer wird, weil in der A–Box schon mehr Informationen vorhanden sind und vom *`Planer`* bei der Erstellung vom Pläner berücksichtigt werden können.
Wird der Instanzenlerner gestartet, werden die von Ihnen einstellbaren Variablen aus der Konfigurationsdatei geladen. Diese Konfigurationsdatei muß von Ihnen vor dem ersten Start des Agenten angelegt werden.

Dateiformat der Konfigurationsdatei

Alle Befehle müssen der Form $< A > = < B >$ entsprechen und in einer Zeile sehen.Zwischen Gross- und Kleinschreibung wird nicht unterschieden.

Die eine zu setzende Variable ist die *deleteTime*, die die Zeitspanne angibt, ab der unbenutzte Instanzen in der globalen A-Box gelöscht werden sollen, damit die A–Box im Laufe der Zeit nicht zu groß wird. Möglich sind Angaben in years, months, weeks, days, hours, minutes, seconds. Die Einheiten können auch, durch Kommata getrennt, kombiniert werden. Wird keine Einheit angegeben, ist days die default–Einheit.
Die andere Variable bezieht sich auf Konzepte, deren Instanzen nicht von den lokalen A–Boxen in die globale übernommen werden sollen. In der Konfigurationsdatei heißen diese Konzepte ***startConcepts***, und sind z.B. die Benutzereingaben, die unter das Konzept searchstring fallen.

Beispiel

Beispielhafte Configdatei des Instanzenlerners

# deleteTime = 8 weeks
# deleteTime = 3 days, 7 hours
deleteTime = 2 months

startConcepts = keyword, searchstring

Kapitel 15

Operatorlerner

Lernst Du wohl,
wirst Du gebratener Hühner voll.
Lernst Du übel,
mußt Du mit der Sau zum Kübel

Martin Luther
Tischreden

Überlegungen zum Operatorlerner

Der Operatorlerner ist **nicht** implementiert!
Im Gegensatz zum Instanzenlerner soll der Operatorlerner nicht nach jeder Suchlanfrage aufgerufen, weil der Operatorlerner die Daten mehrerer Suchanfragen miteinander vergleichen können muß. Deshalb soll er ein eigenständiges Programm sein, das vom Shellbenutzer gestartet werden muß.

Der Operatorlerner soll auch auf dem Planarchiv arbeiten, das die gesammelten Informationen vorangegangener Suchanfragen enthält, und müßte Methoden zur Verfügen haben, um diese Informationen analysieren zu können.

Die eine Idee für den Operatorlerner ist, daß er die Gewichtungen der einzelnen Operatoren neu berechnet, je nachdem wie erfolgreich der Operator in den vorangegangenen Suchen war. Die andere Idee ist, geeignete Reihenfolgen, in denen die Operatoren hintereinander ausgeführt werden sollen, zu finden und sie mit Hilfe der Compound–Operatoren den Planern zu Verfügung zu stellen.

Das Problem ist nun, Kriterien zu bestimmen, mit denen man die Güte von Operatoren berechnen kann. Die Tatsache, daß sie zu Plänen gehört haben, die zum Erfolg geführt haben, ist wahrscheinlich nicht ausreichend, da der Erfolg einer Suchanfrage nicht ausschließlich von den Operatoren abhängt. Bei den Suchmaschinen–Operatoren könnte man die Anzahl der Ergebnisse und deren Konfidenzwert zur Berechnung der Güte heranziehen. Aber auf Grund der unzureichenden Datenmenge, die zur Analyse zur Verfügung standen, war es uns nicht möglich zu fundierten Erkenntnissen zu kommen.

Anhang A

Das Logbuch: die Klasse `LogService`

Es ist von großem Vorteil, die Fehler, aus denen man lernen kann, recht frühzeitig zu machen.
Winston Churchill

Die Klasse `LogService` stellt eine einfache Möglichkeit dar, während eines Programmlaufs Informationen — welche sich typischerweise auf eben jenen beziehen — in Form von Zeichenketten in eine Datei — im folgenden „Logdatei" genannt — zu schreiben. Sinn und Zweck solcher „Logbucheintragungen" sind i. a. das bessere Verständnis von komplexen/unüberschaubaren Programmabläufen und das Lokalisieren von Fehlern im Programm.

Die Klasse `LogService` ist als `abstract` definiert, weil sie ausschließlich statische Methoden, Variablen und Konstanten enthält (sprich Klassenmethoden, -variablen und -konstanten), und eine Instanziierung damit nicht nötig und deshalb auch nicht sinnvoll ist.[1]

Ein Eintrag in die Logdatei muß eine Priorität besitzen, welche sich in der Regel auf die Wichtigkeit der Information beziehen sollte. Sind Logdateieinträge für eine Fehlersuche bestimmt, so kann die Priorität auch als Synonym für eine bestimmte Methode o. ä. "missbraucht" werden. Die Klassenmethode `log`, der als erster Parameter eine Zahl vom Typ `int` und als zweiter Parameter ein *`String`* übergeben wird, nimmt die Eintragung in die Logdatei vor. Dabei ergibt sich der Eintrag aus der Verkettung von der Priorität, dem aktuellen Systemdatum und der eigentlichen Information. Sollte die Logdatei noch nicht bzw. nicht mehr existieren, so wird sie neu erzeugt. **Bei massiver Nutzung dieses Services, zum Beispiel zur Fehlersuche, kann die Logdatei unter Umständen (sehr) schnell (sehr) groß werden, so daß auf rechtzeitiges, manuelles Löschen zu achten ist.** Aus diesem Grund existiert die Klassenvariable `logLevel`. Deren Funktionalität besteht darin, daß Logdateieinträge, die eine geringere Priorität als deren Wert besitzen, nicht an die Logdatei angehangen werden.[2] Es ist somit nicht nötig, Programmcode, der Logdateieinträge zur Fehlersuche erzeugt, zu ändern, wenn diese nicht (mehr) gewünscht werden.[3] Um die Nutzung dieser Funktionalität zu erleichtern, existieren Klassenkonstanten, die als Werte für `logLevel` gedacht sind. Prioritäten, die bei der Fehlersuche eingesetzt werden, sollten aus $[0, 99]$ sein, wenn diese Konstanten benutzt werden. Initialer Wert von `logLevel` ist der

[1] außer zur abkürzenden Schreibweise vielleicht . . .
[2] Sie werden einfach „vergessen", enden also direkt in der „garbage collection".
[3] Die Funktionsaufrufe finden nach wie vor statt, was den Programmablauf dann unnötigerweise verlangsamt.

Wert der Klassenkonstante `DEBUG`. Der Wert von `logLevel` kann auch über die Konfigurationsdatei gesetzt werden.

Neben der Methode `log` gibt es u. a. noch die Methoden `logInfo`, `logWarning`, `logError` und `logFatal`, bei denen die Priorität automatisch auf den Wert der entsprechenden Klassenkonstante gesetzt wird. Ferner existiert zu jeder der genannten Methoden eine überladende Mehode, der zusätzlich ein Objekt übergeben werden muß. Bei diesen Methoden wird vor die übergebene Nachricht der Klassenbezeichner des übergebenen Objekts gesetzt.

Der Name der Logdatei wird durch die Klassenvariable `logfileName` bestimmt, welche mit der Zeichenkette "`agent.log`" initialisiert ist. Über die Konfigurationsdatei (s. II D.1)kann diese Einstellung überschrieben werden. **Sollte ein Fehler[4] bei einer Logdateieintragung auftreten, wird das Programm mit einer entsprechenden Nachricht auf der Standardausgabe abgebrochen.** Für die genauen Signaturen der Methoden und die Namen der Konstanten sei auf die JavaDoc–HTML–Dokumentation der Klasse `LogService` hingewiesen.

[4]genauer: eine IO-Exception

Anhang B

Die Klassifikatoren unterstützende Werkzeuge

B.1 dbExplorerConsole

Das Tool `dbExplorerConsole` ist ein menügesteuertes Programm, mit dem man Datenbanken (`ClassifierDatabase`) erzeugen, anzeigen und bearbeiten kann. Die Entwicklung dieses Tools ist noch nicht abgeschlossen, es ist aber schon eine funktionierende Version vorhanden. Für die Benutzung ist lediglich die Kenntnis der Eigenschaften der `Classifier` wichtig. Diese Eigenschaften können in der Javadoc–Dokumentation zu BotIshelly nachgelesen werden.

B.2 ClassifierGenerator

B.2.1 Einleitung

Mit dem Tool `ClassifierGenerator` sind Sie in der Lage, die Klassifiziererdatenbank auf eine einfache Art und Weise zu erzeugen oder zu erweitern. Die Informationen mit welchen Beispielen und Werten die Klassifizierer in die Datenbank eingefügt werden sollen entnimmt dieses Tool im wesentlichen dem `ClassifierConfigFile`, dass sich im Paket `dataanalysis.tools` befindet. Weiteres zum konkreten Inhalt der Konfigurationsdatei können Sie der Javadoc-Dokumentation entnehmen.

B.2.2 Aufrufsyntax

Das Tool wird mit (mindestens) zwei Parametern und eventuellen Optionen aufgerufen. Aufgerufen wird das Tool wie folgt:

```
java classifiergenerator.ClassifierGenerator [-v] [-f]
        Databasefilename Classifiername...
```

- **Parameter Databasefilename**
  Hier geben Sie den Dateinamen der `ClassifierDatabase` an, in der die Klassifikatoren gespeichert werden sollen. Existiert unter diesem Dateinamen keine `ClassifierDatabase`, so wird eine neue `ClassifierDatabase` erzeugt.

- **Parameter Classifiername...**
  Hier geben Sie den oder die Namen der Klassifizierer an, die erzeugt werden sollen. Die Endung „.examples“ wird automatisch entfernt, d.h. durch den Parameter „test.examples“ wird ein Klassifizierer mit dem Namen „test“ erzeugt. Sie können also immer die Dateinamen der Konfigurationsdateien verwenden, wenn die Konfigurationsdateien die Endung „.examples“ besitzen.

- **Option -v**
  Wird diese Option angegeben, dann wird der Inhalt der `ClassifierDatabase` nach dem Abarbeiten aller Klassifizierer auf dem Bildschirm ausgegeben. Ansonsten wird nur eine Zusammenfassung angezeigt, wie viele Klassifizierer hinzugefügt wurden.

- **Option -f**
  Wird diese Option angegeben, dann werden eventuell schon in der `ClassifierDatabase` vorhandene Klassifizierer gleichen Namens ersetzt.

B.2.3 Aufrufbeispiele

- `java classifiergenerator.ClassifierGenerator -v -f`
  `testDB.cldb productpage.examples`
  Es wird eine `ClassifierDatabase` mit dem Namen „testDB.cldb“ verwendet (ggf. erzeugt). In dieser Datenbank wird ein Klassifizierer mit dem Namen „productpage“ erzeugt, wobei ein eventuell schon vorhandene Klassifizierer mit diesem Namen ersetzt wird. Die Daten zur Erzeugung werden aus der Datei „productpage.examples“ verwendet. Am Ende wird eine Zusammenfassung und der Inhalt der Datenbank angezeigt.

- `java classifiergenerator.ClassifierGenerator`
  `testDB.cldb test.examples foo.examples`
  Es wird eine `ClassifierDatabase` mit dem Namen „testDB.cldb“ verwendet (ggf. erzeugt). In dieser Datenbank werden zwei Klassifizierer erzeugt. Sollte z.B. schon ein Klassifizierer mit dem Namen „test“ vorhanden sein, wird eine Fehlermeldung ausgegeben und mit dem nächsten Klassifizierer fortgefahren. Am Ende wird nur die Zusammenfassung angezeigt.

B.3 GetLocalExamples

B.3.1 Einleitung

Mit dem Tool `GetLocalExamples` sind Sie in der Lage, die Beispiele für die Klassifikatoren Ihres Agenten auf dem lokal verfügbaren Speicherplatz (z.B. Festplatte) abzulegen.

Durch diese Maßnahme wird das spätere Training der Klassifikatoren erheblich beschleunigt und der Netzverkehr bei der (empirischen) Einstellung der Klassifikatorparameter erheblich gesenkt. Desweiteren können so Testläufe mit dem Tool `ClassifierTester` innerhalb einer kurzen Zeit durchgeführt werden.

B.3.2 Benutzung

Dieses Tool erzeugt jeweils aus einem gegebenen `ClassifierConfigFile` eine lokalisierte Version, die im Unterverzeichnis „lokal“ abgespeichert wird. Die Beispiele in der Datei werden in einem Unterverzeichnis abgespeichert. Geben Sie z.B. eine Datei „test1.examples“ vor, so wird im Unterverzeichnis „lokal“ ein Unterverzeichnis „test1“ erstellt, in dem die Beispiele aus dem `ClassifierConfigFile` abgespeichert werden. Das angepasste `ClassifierConfigFile` wird im

Unterverzeichnis „lokal“ abgespeichert. Sollte dieses schon vorhanden sein, so werden nur die Beispiele geholt, die noch nicht lokal vorhanden sind. Es ist also einfach möglich eventuelle Serverausfälle zu überbrücken, indem Sie mehrere Durchläufe zu verschiedenen Zeiten machen.

B.3.3 Aufrufsyntax

Das Tool wird mit einem Parameter und eventuellen Optionen aufgerufen. Aufgerufen wird das Tool wie folgt:

```
java classifiergenerator.GetLocalExamples [-r] [-v] [-a]
        [-n<num>] ConfigFilename
```

- **Parameter ConfigFilename**
  Hier geben Sie den Dateinamen des `ClassifierConfigFile` an, dessen Beispiele geholt werden sollen.

- **Option -r**
  Wird diese Option angegeben, so wird nicht der formatierte Text, sondern der Quelltext der Beispiele (also z.B. HTML-Code) abgespeichert. Bei gesetzter Option wird die Methode `getRaw()` der jeweiligen ***NetEntity*** aufgerufen, ansonsten die Methode `getText()`.

- **Option -v**
  Wird diese Option angegeben, so werden detaillierte Ausgaben auf dem Bildschirm ausgegeben. Jede Beispiel–URL wird dann ausgegeben.

- **Option -a**
  Wird diese Option angegeben, so wird nicht automatisch „.html“ an jede geholte Beispiel–URL angehangen. Zur Zeit ist das Anhängen von „.html“ notwendig, da die Beispiele nur so wieder verarbeitet werden können (bei file–URL's muss die Endung „.html“ vorliegen, um `HTMLTextNetResults` zu erhalten.).

- **Option -n<num>**
  Wird diese Option angegeben, so werden immer **num** Beispiele gleichzeitig geholt. Dadurch wird die Geschwindigkeit im Gegensatz zum sequentiellen Abruf erheblich erhöht. Als praktikabel hat sich ein Wert von 20 herausgestellt.

Anhang C

Generierung von Operatoren

C.1 OperatorGenerator

Dieses Werkzeigt ermöglicht es dem Shellbenutzer schnell und einfach Operatorcoderahmen zu erzeugen. Das lästige und fehleranfällige erzeugen der Konstruktoren entfällt dadurch.

Das Werkzeug wird mit folgender Kommandozeile gestartet:

```
java operatorgenerator/OperatorTextFileReader <Operator-Signaturdatei>
```

Das Werkzeug legt dann für jeden in der Operator–Signaturdatei beschriebenen Operatoren eine Datei <Operatorname>.java an. Bestehende Dateien gleichen Namens werden *ohne* Rückfrage *überschrieben.*

Die Operator–Signaturdatei muß dabei folgendes Format haben:

Jeder neue Operatorabschnitt wird mit der Zeile @name <Operatorname> Danach folgen die Vorbedingungsprädikate, die zeilenweise eingeleitet durch @precondition aufgezählt werden. Anschliessend wird mit @postcondition ein Nachbedingungsabschnitt eingeleitet. Nach dieser Zeile folgen die zu dieser Nachbedingung gehörenden Add- und Deletelisteneinträge jeweils zeilenweise mit den Tags @addlistitem bzw. @deletelistitem eingeleitet. Eine weitere Nachbedingung wird wieder mit dem @postcondition-Tag begonnen. Zeilen die mit einem # beginnen werden ignoriert.

Hinter den Add- oder Deletelistenprädikaten sind durch einen „:" abgetrennt die Buchstaben „y" oder „n" je Prädikatsargument einzutragen.„y" bedeutet, daß diese Argument neu vom Operator erzeugt wird und daher vom Planer als dynamisches Weltobjekt behandelt werden muß. Die genaue Syntax entnehmen Sie bitte Anhang II D.4.
Beispiel:

```
# blubber-Operator
@name blubber
@precondition be_involved_with(a,b)
@precondition archive_url(a)
@postcondition
@addlistitem has_downloadurl(a,b):n,n
@addlistitem distribution(a):n
@addlistitem contains_software(a,b):n,n
@deletelistitem be_involved_with(a,b):n,n
@deletelistitem belongs_to(a,b):n,n
```

Der blubber–Operator ist also ein Operator mit dem Namen *blubber*, zwei Vorbedingungsprädikaten und genau einer Nachbedingung. Die Nachbedingung besteht einer Addliste mit drei Einträgen und einer Deleteliste mit zwei Einträgen.

Zu beachten ist, daß Prädikatsargumente mit gleichem Namen im Konstruktor denselben ***PredArguments*** entsprechen. Das bedeutet, im obigen Beispiel, daß nur genau zwei ***PredArguments*** erzeugt werden: das ***PredArgument*** mit dem Namen a und das ***PredArgument*** mit dem Namen b. Natürlich macht es keinen Sinn in einer Nachbedingung ein Prädikatargument mit dem Namen eines Prädikatarguments aus der Vorbedingung erzeugen zu wollen.

Abweichend von der in Anhang II D.4 genannten Syntax können bei Vorbedingung die Kennzeichen `:n` oder `:n,n` bzw. `:n` oder `:n,n` entfallen, da sie hier bedeutungslos sind. Vorbedingung können keine Objekte erzeugen.

Fehlerhafte Zeilen in der Operator–Signaturdatei ignoriert das Hilfsprogramm unter Angabe der fehlerhaften Zeilennummmer.

C.1.1 grafische Oberfläche

Es gibt eine prototypische Implementation eines grafischen Frontends zu dem beschriebenen Operatorgenerator. Dieses ist aber weder benutzerfreundlich noch komplett fertig gestellt. Dem experimentierfreudigen Leser wollen wir aber die Möglichkeit nicht vorenthalten. Fertig kompiliert kann man es mit

```
java operatorgenerator <A-Box-TextDatei>
```

starten. In diesem Tool können dann Operatoren zusammengeklickt werden. Da die Arbeiten an diesem Frontend nicht abgeschlossen wurden und es nicht fester Bestandteil von BotIshelly ist, wird für eine weitere Dokumentation auf die Sourcecode–Dokumentation verwiesen.

Anhang D

Beschreibungssprachen und Dateiformate

> Neuerdings habe ich mir's zur Richtschnur gemacht: In Sachen, die ich nicht verstehe, und es tut einer etwas, das ich nicht begreife, so macht er's dumm und greift's ungeschickt an; denn das, was schicklich und recht ist, begreift man auch in unbekannten Dingen; wenigstens muß es einer einem leicht und bald erklären können
>
> *Goethe, an Charlotte von Stein*
> 12.9.1780

D.1 Konfigurationsdatei des Agenten

In der Konfigurationsdatei stehen alle wichtigen Daten, die für eine Suche notwendig sind.

Die von uns zu Verfügung gestellte Konfigurationsdatei heißt config.sys.

Falls Sie eine andere Konfigurationsdatei benutzen möchten, müssen Sie darauf achten, daß Sie in der Klasse `systemcontrol.Flowcontrol` den Namen der Konfigurationsdatei ändern. Folgendes müssen Sie unbedingt beachten: Kommentare werden mit `#` eingeleitet und dürfen nur am Anfang einer Zeile stehen. Abkürzungen wie $HOME bitte NICHT verwenden. Außerdem ist die Datei case sensitiv.

Hier ist ein Beispiel für eine Konfigurationsdatei.

```
# config.sys Version 0.2

# Globale Konfigurationsdatei fuer BotIShelly

# Working Directory wird vorangestellt, wenn bei Pfadangaben ein relativer
# Pfad angegeben wird

Working Directory = /home/pg343/schroet3
```

Falls Sie relative Pfade benutzen, sollten Sie das beim Laden der Daten in der `Flowcontrol` beachten und diese ergänzen. Zurück zu unserem Beispiel:

```
# relative oder absolute Pfade der zu ladenden Dateien

A_BoxDatei = sourcen/integration-shell/DebianSuchagent.abox
A_BoxObject = sourcen/integration-shell/DebianSuchagent.aboxObject
T_Box = sourcen/integration-shell/DebianSuchagent.tbox
OperatorDBDatei = sourcen/integration-shell/OperatorDB.opdb
OperatorDBObject = sourcen/integration-shell/OperatorDB.opdbObject
ClassDatabase = /home/pg343/share/DebianSuchagent/DebianSuchagent.cldb

# Bool der definiert, ob die Classifier DB am Ende der Suche
gespeichert werden soll, oder nicht

SaveClassDB = false

# Einstellungen fuer die Log-Datei

# wird LogPath nicht angegeben, wird der Default benutzt ($HOME/agent.log)
logfileName = agent.log

# wird LogLevel nicht angegeben, wird der LogLevel auf 0 gesetzt
logLevel = 0

# Proxy Settings

proxySet  = true
proxyHost = fbi-www
proxyPort = 3128

# maximale Anzahl der Instanzen pro Operatorvorbedingungspr"adikat bestimmen,
die bei der Suche ber"ucksichtigt werden

MaxInstance = 10
```

Es darf in Ihrer Konfigurationsdatei nichts davon fehlen, was Sie in unserem Beispiel sehen. Natürlich können Sie diese Datei beliebig erweitern, sofern Sie die syntaktische Einsränkungen beachten.

D.2 Konfigurationsdatei für die Operatorendatenbank

Die Konfigurationsdatei für die Operatorendatenbank ist vielleicht die schlichteste und am wenigsten benutzerfreundliche Konfigurationsdatei in BotIshelly. Dafür wird sie aber auch nur einmal im gesamten Agentenleben gebraucht. Ihre Aufgabe ist es die Operatorendatenbank mit den geschriebenen Operatoren zu füllen. Die Syntax dieser Datei ist einfach:

In der ersten Zeile steht der Name der Operatorklasse einschließlich des Paketnamens, in dem er sich befindet. In der darauffolgenden Zeile steht die Gewichtung dieses Operators mit der er initial in die Datenbank eingehen soll. Die Gewichtung ist ein double-Wert.

Kommentare und Leerzeile sind in dieser Datei nicht erlaubt.

Fehlerhafte Zeilen werden ignoriert und ihre Zeilennummer im Agentenlog angegeben.

D.3 Syntax für ClassifierConfigFile

D.3.1 Regeln

Die folgenden Regeln gelten für das `ClassifierConfigFile`:

- Führende Leerzeichen in einer Zeile werden entfernt.
- Tab-Zeichen werden in Leerzeichen umgewandelt.
- Leerzeilen werden entfernt.
- Beispiele die Leerzeichen enthalten sind in Anführungszeichen (“) einzuschließen.
- Kommentare werden durch das Zeichen „#“ als erstes Zeichen der Zeile eingeleitet.
- Soll ein Beispiel „asis“ an den Klassifizierer gegeben werden, so muss das erste Zeichen in der Zeile ein „@“ sein.
- Schlüsselwörter beginnen mit dem Kommentarzeichen gefolgt von beliebig vielen Leerzeichen und anschließend dem Schlüsselwort. Das mögliche Argument wird durch ein „=“–Zeichen vom Schlüsselwort getrennt.
- Folgt ein `double`–Wert auf das Beispiel (durch Leerzeichen getrennt), so wird dieser als Bewertung des Beispiels verwendet, ansonsten der Wert, der durch das Schlüsselwort `DEFAULT` festgelegt wurde.

D.3.2 Schlüsselwörter

Es gibt die folgenden Schlüsselwörter (Groß–/Kleinschreibung wird unterschieden!):

- `CREATED` : gibt das Erstellungsdatum der Datei an.
- `DEFAULT=f` : ein `double`–Wert, der als Standard–Klassifikation benutzt wird. Ist f kein gültiger `double`–Wert, so wird dieser ignoriert. Standardwert ist 1.0.
- `END` : bricht das Einlesen nach dieser Zeile ab.
- `NEEDS=x` : bezeichnet einen Namen eines benötigten Klassifizierers. Wird noch nicht beachtet.
- `NEGATIVDICT=x` : den Namen des Negativ–Wörterbuches (Stopwortliste) des Klassifizierers. Standardwert ist „Global_Stoplist“.
- `THRESHOLD=f` : ein `double`–Wert, der den Threshold des Klassifizierers festlegt. Standardwert ist 0.5.
- `TYPE=x` : legt den Typ des Klassifizierers fest (Klassenname).
- `VECTORIZER=x` : den Namen des Vektorisierers des Klassifizierers.
  Standardwert ist „<Klassifizierername>_Vectorizer“.
- `WORDDICT=x` : den Namen des Feature–Wörterbuches des Klassifizierers. Durch dieses Standardwert ist „<Klassifizierername>_Words“.

D.3.3 Beispiel für ein ClassifierConfigFile

```
#  Ein Beispiel
#  TYPE=dataanalysis.classifiers.DummyClassifier
#  DEFAULT=1.0
#  THRESHOLD=0.5
#  NEEDS=a_classifier_name
#  VECTORIZER=a_classifier_name_Vectorizer
#  WORDDICT=a_classifier_name_Features
#  NEGATIVDICT=Global_Stoplist
#
@"ein String als Beispiel" 0.75
# Beispiel-URL mit Benutzung des DEFAULT-Wertes
http://localhost/
#
# Ende der Datei
# END
```

D.4 Operator–Signaturdateisyntax

Was ein <gültiger Java-Variablenbezeichner> ist entnehmen sie bitte der Java-Dokumentation. Was eine <Zeichenkette> oder ein <Zeilenumbruch> ist sollte dem Leser bekannt sein. Die <Operatorsignaturdatei> soll als Textfile geschrieben werden.

```
<Name>                   ::=<gültiger Java-Variablenbezeichner>
<Kommentar>              ::=#<Zeichenkette><Zeilenumbruch>|
                            <Kommentar><Kommentar>
<neu erzeugt>            ::=n|y
<einstelliges Prädikat>  ::=<Name>(<Name>):<neu erzeugt>
<zweistelliges Prädikat>::=<Name>(<Name>,<Name>):<neu erzeugt>,<neu erzeugt>
<Prädikat>               ::=<zweistelliges Prädikat>|
                            <einstelliges Prädikat>
<Vorbedingung>           ::=[@precondition <prädikat>]|
                            <Vorbedingung><Vorbedingung>|
                            <Kommentar><Vorbedingung>
<Addlisteneintrag>       ::=@addlistitem <prädikat>|
                            <Kommentar><Addlisteneintrag>
<Deletelisteneintrag>    ::=@deletelisteitem <prädikat>|
                            <Kommentar><Deletelisteneintrag>
<Nachbedingung>          ::=@postcondition {<Zeilenumbruch><Addlisteneintrag>}
                                           {<Zeilenumbruch><Deletelisteneintrag>}|
                            <Nachbedingung><Zeilenumbruch><Nachbedingung>|
                            <Kommentar><Nachbedingung>
<Operatorname>           ::=@name <Name>|
                            <Kommentar><Operatorname>
<Operatorsignatur>       ::=<Operatorname><Zeilenumbruch><Vorbedingung>
                                          <Zeilenumbruch><Nachbedingung>|
                            <Kommentar><Operatorsignatur>
<Operatorsignaturdatei>  ::=<Operatorsignatur>{<Zeilenumbruch><Operatorsignatur>}
```

Index

Teil III

Benutzungshandbuch Firmenagent

Inhaltsangabe Teil III

Kapitel 1

Einleitung

1.1 Aufgabenstellung und Motivation

Gegenstand der Projektgruppe 343 war die Implementation einer Bibliothek, mit deren Komponenten die Implementation eines Agenten, auch Softbot genannt, wesentlich vereinfacht werden sollte. Nachdem diese Bibliothek Namens BotIshelly fertiggestellt war, bestand die Aufgabe der Projektgruppe nun darin, an einem Beispiel zu zeigen, daß BotIshelly auch das Geforderte leistet. Die Projektgruppe beschloß, zur Demonstration der Bibliotheksfunktionalität einen Agenten mit BotIshelly zu implementieren, der Homepages von Firmen im WWW findet, und zwar ohne das übliche "Durchklicken" durch Unmengen von Suchmaschinenergebnissen und das Ausprobieren von ad hoc gebildeten URLs. Genau diese immer gleichen Vorgehensweisen können nämlich automatisiert werden und werden auch von dem Agenten angewandt.

Das Hauptziel bei der Entwicklung des hier beschrieben Agenten war somit nicht eine bestmögliche Performanz, sondern vielmehr war im zeitlich begrenzten Rahmen der Projektgruppe die Demonstration der Fuktionalität der Bibliothek BotIshelly das Ziel. Damit ist die korrekte Funktion der von BotIshelly bereitgestellten Einzelkomponenten, wie z. B. die der Planer, die der Klassifikatoren oder auch die der Datenbeschaffung gemeint, und vor allem auch das reibungslose Zusammenspiel dieser, wie es z. B. in den Klassen der Systemsteuerung nötig ist.

1.2 Realisierung

Realisiert wurde der Firmen–Homepage–Agent als Javaservlet, da in BotIshelly schon Unterstützung für diese Lösung enthalten ist. Außerdem ermöglicht ein Servlet, den Agenten auf einem schnellen Rechner mit schnellem und permanentem Netzzugang auszuführen und seine Bedienung bzw. Steuerung von einem anderen Rechner aus zu tätigen. Darüberhinaus kann ein Agent durch diese Architektur mehrere Anfragen gleichzeitig bearbeiten[1]. Die hauptsächliche Arbeit bei der Realisierung bestand darin, problemspezifische Operatoren zu implementieren. Vorher waren jedoch die Signaturen und damit deren jeweilige Funktionalität festzulegen. So gibt es z. B. einen Operator, der aus der URL einer HTLM–Seite, die für eine Seite über ein bestimmtes Produkt gehalten wird, eine neue URL bildet, für die er dann testet, ob es sich um eine Firmenhomepage handelt. Da die Pläne des Agenten maßgeblich von den Operatorensignaturen bestimmt werden, ist hier sehr viel Abstimmungsarbeit zu leisten. Die Performanz des Agenten steht und fällt mit der Menge der Operatoren, für die man sich entscheidet. Bei der Entwicklung dieses Agenten wurde die anfänglich angedachte Menge an relativ einfachen Operatoren durch eine kleinere Menge

[1] wobei die Systemressourcen dem Parallelitätsgrad schnell ein jähes Ende setzen

komplexerer Operatoren ersetzt, da sonst die Anzahl der möglichen Pläne "auszuufern" drohte. Das Potential des Agenten ist an dierser Stelle sicher nicht (voll) ausgeschöpft worden. Eine genauere Beschreibung der realisierten Operatoren folgt später im Text. Nun soll jedoch beschrieben werden, wie der Agent benutzt werden kann.

Kapitel 2

Benutzung des Agenten

2.1 Auf die Plätze, ...

In diesem Abschnitt wird erklärt, was alles benötigt wird, um den Firmen–Homepage–Agenten "laufen" zu lassen.

Zunächst wird die Bibliothek BotIshelly benötigt, welche wiederum das JDK 1.2.1 (Solaris 2) benutzt. Daneben werden noch die agentenspezifischen Javaklassen benötigt. Da der Firmen–Homepage–Agent als Javaservlet implementiert ist, wird ein Programm benötigt, welches ein solches ausführen kann. Getestet wurde der Agent mit dem Servletrunner des JSDK der Version 2.0 (Solaris 2). Nicht zu vergessen sind die von BotIshelly benötigten, aber nicht enthaltenen Javaklassen bzw. –bibliotheken[1]. Schließlich ist noch ein Browser, z. B. Netscapes Communicator, mit dem auch getestet wurde, nötig, um den Agenten zu steuern.

2.2 ... fertig, ...

In diesem Abschnitt wird gezeigt, wie die im vorigen Abschnitt beschrieben "Einzelteile" benutzt werden, um den Firmen–Homepage–Agenten zu starten.

Im Agentenverzeichnis befindet sich u. a. die Datei `config.sys` (s. II D.1), in der Einstellungen für das Verhalten des Agenten vorgenommen werden. Hier müssen Sie mindestens den Pfad des Arbeitsverzeichnisses `Working Directory` auf das Agentenverzeichnis setzen. Desweiteren müssen Sie den Pfad `ClassDatabase` auf die gewünschte Klassifizierer-Datenbank setzen. Eine Klassifizierer Datenbank für den Firmen-Suchagenten finden Sie in `share/FirmenSuchagent/FirmenSuchagent.cldb`.

Wird der Servletrunner des JSDK 2.0 benutzt, müssen weitere Einstellungen in der Datei *Agentenverzeichnis*`/io/servlet.properties` vorgenommen werden. Die Javaklasse, die der Java–Servlet–Runner ausführen muß, ist `/io/InputmaskServlet` im Agentenverzeichnis. Dies entspricht dem Wert der Variable `servlet.agent.code` in `/io/servlet.properties`. Der Servletrunner muß dem Konstruktor dieser Klasse beim Starten das Argument `configfile=`*Agentenverzeichnis*`/config.sys` übergeben (Variable `servlet.agent.initArgs`).

Als letztes müssen Sie noch die Umgebungsvariable `CLASSPATH`, die dem Javainterpreter angibt, wo er die benötigten Klassen suchen soll, richtig setzen. U. a. müssen sich das Verzeichnis, in dem BotiShelly installiert ist, das Agentenverzeichnis, das JDK und das Verzeichnis des Servletrunner (JSDK) im Classpath befinden. **Dabei muß das Agenten- vor dem BotIshelly–Verzeichnis stehen.**

[1] siehe dazu Teil II, BotIshelly–Handbuch

Gestartet wird der Servletrunner mit `servletrunner -d` *Agentenverzeichnis*`/io` aus dem Verzeichnis *Agentenverzeichnis*`/io`. Sie können zum Starten auch den Befehl **`make newrun`**, ausgeführt im Agentenverzeichnis, benutzen.

2.3 ...los!

In diesem Abschnitt wird schließlich anhand eines Beispiels beschrieben, wie mit dem laufenden Agenten die Homepage einer bestimmten Firma gesucht — und hoffentlich auch gefunden — wird.

Nachdem das Agentenservlet mit Hilfe des Servletrunners wie oben beschrieben gestartet wurde, kann nun mit einem Browser die Eingabeseite des Agenten geladen werden. Deren URL wird typischerweise durch Konkatenation

- des Protokolls (“`http:`”),
- des Rechnernames (hier “`//herz`”),
- des Ports, über den der Servletrunner zu erreichen ist, (hier “`:8080`”) und
- des Identifikator des Agenten im Servletrunner (hier “`/servlet/agent`”) gebildet.

Nachdem der Browser die Seite mit der Eingabemaske geladen hat, kann durch die Eingabe bestimmter Daten der gesuchten Firma die Suche gestartet werden. Zur Veranschaulichung siehe den Screenshot der Eingabemaske.

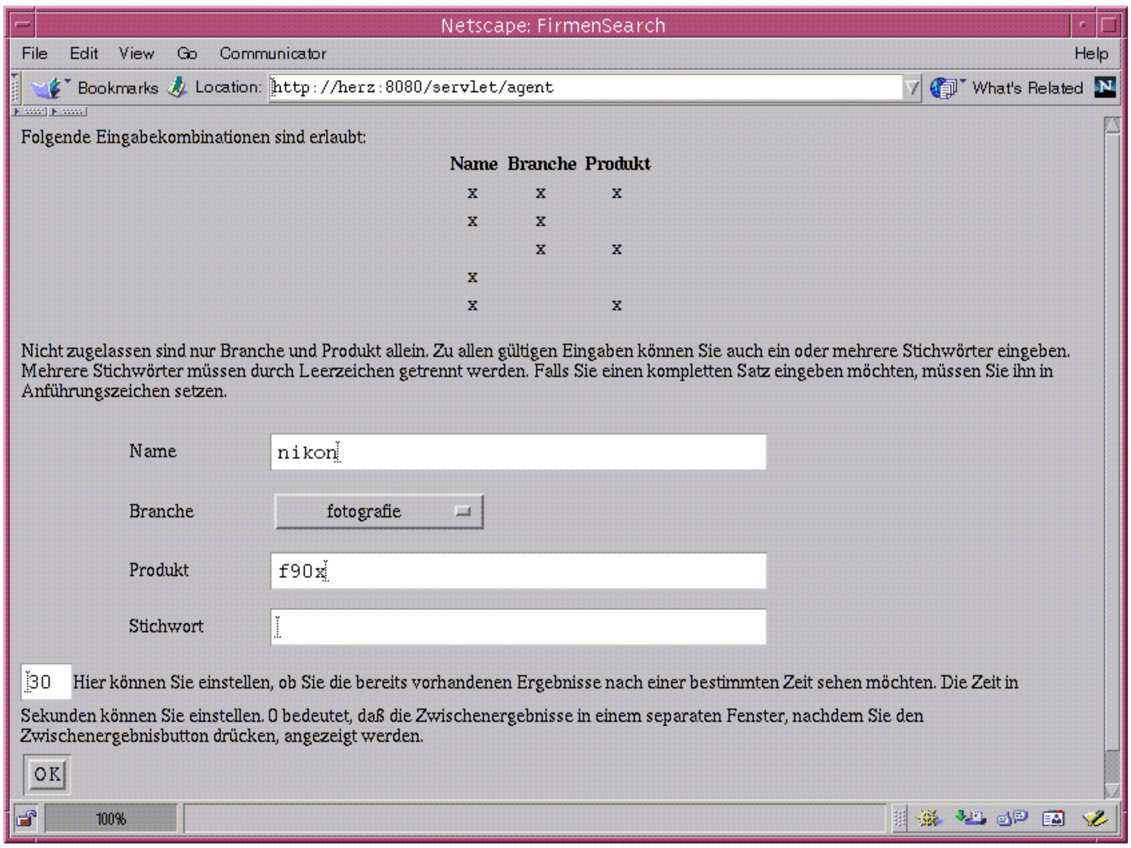


Abbildung 2.1: Eingabemaske des Firmen–Homepage–Agenten

Im oberen Teil der Eingabemaske werden die zulässigen Kombinationen an Firmeninformationen (Name, Branche, Produkt) beschrieben, für die eine Suche möglich ist. Die Branche kann dabei aus einer Liste ausgewählt werden, wobei auch die Wahl "alle Brachen" möglich ist. Name und Produkt der Firma sind Freitexteingaben. Wird ein Produkt angegeben, so werden nur dann Firmen–Home–Pages ausgegeben, wenn eine passende Produktseite gefunden wird. Im allgemeinen wird entweder versucht, von einer Firmenseite durch "Spidern" eine passende Produktseite zu finden, oder aber ausgehend von einer Produktseite, z. B. durch das Folgen von "Home–Links" oder das "Abschneiden der URL", die passende Firmenseite. Die jeweiligen "Startseiten" werden dabei durch das Befragen von Suchmaschinen gefunden. Wird eine Branche angegeben, so wird bei der Suchmaschine *Yahoo!* in der entsprechenden Kategorie gesucht und bei anderen Suchmaschinen die Anfrage durch Synonyme ergänzt. Wahlweise können zusätzlich Stichworte angegeben werden, die die Suche dadurch einschränken, daß nur Firmen–Home–Pages gesucht werden, auf denen diese Worte vorkommen.

Schließlich kann noch eingegeben werden, in welchem Intervall die Ergebnisausgabe aktualisiert werden soll. Nachdem die Daten der Suche eingegeben wurden, wird die Suche durch klicken des OK–Buttons gestartet. Dadurch wird der Browser auch veranlaßt, die Seite, auf der die Zwischenergebnisse angezeigt werden, zu laden. Ein Beispiel ist in Form eines weiteren Screenshots gegeben.

Abbildung 2.2: Zwischenergebnismaske des Firmen–Homepage–Agenten

Diese Zwischenergebnisseite wird nun periodisch in dem zuvor angegebenem Intervall aktualisiert. Um den angegeben Links zu folgen, ist es günstig, die bei den meisten Browsern vorhandene Funktion "Öffne Link in neuem Fenster" zu verwenden. Um die Suche vorzeitig zu beenden, kann — und sollte — der ABORT–Button der Zwischenergebnismaske verwendet werden. Anderenfalls wird das Ende der Suche angezeigt werden.

Kapitel 3

Hinter den Kulissen

3.1 Die Ziele des Planers

Die Pläne, die der Firmenagent verfolgt, bestehen aus Operatoren, die von den Planern in eine sinnvolle Reihenfolge gebracht wurden. „Sinnvoll" heißt dabei, daß nach Abarbeitung der Folge von Operatoren (also des Plans) durch den Planausführer die Zielprädikate erfüllt sein sollen. Um das Ziel, eine Homepage einer Firma zu finden, erreichen zu können, muß der Plan eine Teilmenge der folgenden Prädikate (vgl. I 5) erfüllen:

- `has_compage (companyname, compage)`
- `contains_keyword (compage, keyword)`
- `contains_productname (prodpage, prodname)`
- `has_productpage (companyname, prodpage)`

Um die Suche zu starten, haben wir fünf mögliche Eingabekombinationen von Firmenname, Branche und Produktname zugelassen, die optional durch ein Stichwort ergänzt werden können. Die erlaubten Kombinationen sind:

- Name, Branche, Produkt
- Name, Branche
- Branche, Produkt
- Name
- Name, Produkt

Beispiel: Wird als Eingabe die Kombination Name = *nikon*, Branche = *fotografie* und Produkt = *f90x* gewählt, so ist das Ziel, die Homepage der in der Fotobranche tätigen und das Produkt f90x vertreibenden Firma Nikon zu finden. Daraus folgt, daß drei der oben genannten Zielprädikate erfüllt sein müssen, nämlich

- `has_compage ("nikon", compage)`
- `contains_productname (prodpage, "f90x")`
- `has_productpage ("nikon", prodpage)`

3.2 Die Operatoren als Trittsteine zum Ziel

Von einer gegebenen Anzahl von Prädikaten aus muß der Planer die Operatoren geschickt hintereinandersetzen, damit, ein Operator nach dem anderen, das Ziel erreicht werden kann. Bei der Umsetzung des Firmenagenten haben wir uns für Operatoren entschieden, die sich in fünf Kategorien einteilen lassen:

1. Operatoren, die sich durch Suchmaschinenanfragen eine Menge von URLs besorgen
2. Operatoren, die von einer einzelnen Adresse aus eine beschränkte Tiefensuche durchführen (Spider–Operatoren)
3. Operatoren, die aus einer vorhandenen URL eine weitere raten
4. Operatoren, die aus einer vorhandenen Seite Informationen zu extrahieren suchen
5. Operatoren, die das Vorkommen eines Wortes auf einer Seite abtesten

Im Folgenden werden die zu den einzelnen Kategorien gehörenden Operatoren kurz vorgestellt. Dabei bezeichne **V** eine Vorbedingung, **N** die erfolgreiche Nachbedingung (unsere Operatoren haben nur eine Nachbedingung) und ein + vor einem Prädikat– oder Prädikatargumentnamen ein Erzeugen in der Add–Liste der Nachbedingung.

3.2.1 Operatoren zur Suchmaschinenanfrage

Diese Operatoren haben gemeinsam, daß sie Suchmaschinen abfragen und bei Erfolg Webadressen zurückliefern, die von den Operatoren abhängende Eigenschaften haben sollen. Im Firmenagenten gibt es drei Operatoren dieser Kategorie, die jeweils für die Suchmaschinen Altavista, Nathan und Yahoo ausprogrammiert sind.

`FindComPageOperator`

V: `in_trade (comname, trade)`
`relates_to (synonym, trade)`
N: `+ has_compage (comname, + compage)`

Aufgabe: Anfrage an eine Suchmaschine mit Hilfe von Firmennamen, Branche und zur Branche passendem Stichwort. Die URLs, die die Operatoren liefern, sollen Firmenseiten sein.

`FindContainsProductOperator`

V: `relates_to (synonym, trade)`
`productname (prodname)`
N: `+ contains_productname (+ productpage, prodname)`

Aufgabe: Ähnlich dem vorangegangenen Operator, nur daß hier nach einer Produktseite gesucht wird. Dabei ist der Firmenname nicht bekannt.

`FindProdPageOperator`

V: `in_trade (comname, trade)`
`deals_in (comname, prodname)`
`relates_to (synonym, trade)`
N: `+ has_productpage (comname, + prodpage)`
`+ contains_productname (prodpage, prodname)`

Aufgabe: Sucht wie der vorherige Operator nach Produktseiten, wobei hier andere Vorbedingungsprädikate einfließen.

3.2.2 Spider–Operatoren

Spider–Operatoren führen von einer Webseite aus eine Tiefensuche durch. Dabei soll eine Unterseite gefunden werden, die gewisse Eigenschaften aufweist. Da diese Suche unter Umständen lange dauern kann, sollte man die Suchtiefe beschränken.

`FindProdPageFromHomeOperator`

V: `has_compage (comname, compage)`
`productname (prodname)`
N: `+ has_productpage (comname, + prodpage)`
`+ contains_productname (prodpage, prodname)`

Aufgabe: Von einer Firmenseite aus wird nach einer Produktseite gesucht, die einen gegebenen Produktnamen enthalten soll. Dabei wird nur auf dem gegebenen Server gesucht, bis zu zwei Linkebenen in die Tiefe.

3.2.3 Aus URLs URLs ratende Operatoren

Häufig kann aus einer gegebenen URL eine andere URL geraten werden. So kann mit ziemlicher Sicherheit aus einer Adresse http://www.bla.de/verkauf/zitronen.html geraten werden, daß die Homepage unter http://www.bla.de zu finden ist. Diese Aufgabe haben die folgenden Operatoren:

`GuessCompageOperator`

V: `in_trade (comname, trade)`
N: `+ has_compage (comname, + compage)`

Aufgabe: Es wird versucht, die Homepage einer Firma zu raten. Die vier programmierten Operatoren raten die Adressen *www.Bla.de*, *www.Bla.com*, *www.Blacom.de* und *www.Bla-com.de* zu einem Firmennamen.

`GuessCompageFromProdpageOperator`

V: `trade (trade)`
`has_productpage (comname, prodpage)`
N: `+ has_compage (comname, + compage)`
`+ in_trade (comname, trade)`

Aufgabe: Die beiden Operatoren `GuessCompageFromProdpageCutOperator` und `GuessCompageFromProdpageHomeOperator` versuchen beide, die Homepage einer Firma zu finden. Der erste Operator schneidet einfach die Host–URL aus der Adresse der Produktseite heraus, während der zweite nach einem Link auf die Home–Seite innerhalb des HTML–Kodes sucht.

3.2.4 Aus Seiten extrahierende Operatoren

Diese Kategorie Operatoren soll auf einer Seite nach Informationen suchen, z. B. dem (noch nicht bekannten) Namen einer Firma.

`GuessCompanynameOperator`

V: `contains_productname (prodpage, prodname)`
N: `+ has_productpage (+ comname, prodpage)`
`+ deals_in (comname, prodname)`

Aufgabe: Dieser Operator versucht, aus der übergebenen Produktseite den Namen der herstellenden Firma zu gewinnen. Die ausprogrammierte Strategie ist eher simpel und beschränkt sich darauf, den Namen der Firma aus der URL der Produktseite herauszuraten und diesen Namen zur Bestätigung auf der Seite zu suchen.

3.2.5 Überprüfende Operatoren

Überprüfende Operatoren durchsuchen eine Seite nach einem bestimmten Wort. Je nach Ausprogrammierung des Operators kann diese Suche weit über einen reinen Stringvergleich hinausgehen. So könnte der Satz „Bernd aß Bananen" durchaus als das Verb „essen" enthaltend klassifiziert werden.

`CompageContainsKeywordOperator`

V: `keyword (key)`
`has_compage (comname, compage)`
N: `+ contains_keyword (compage, key)`

Aufgabe: Der für den Firmenagenten programmierte Operator überprüft, ob das übergebene Wort als String irgendwo auf der Seite vorkommt. Groß–/Kleinschreibung wird ignoriert.

3.3 Funktioniert das?

Abschließend sei hier gezeigt, daß mit den von uns gewählten Operatoren tatsächlich Ziele erreicht werden können. Dazu führen wir hier das Beispiel aus III 3.1 weiter fort. Gegeben haben wir eine mit Synonymen zu den einzelnen Branchen gefüllte A–Box sowie eine Reihe von instantiierten oder halbinstantiierten Prädikaten. Dazu zählen natürlich die Zielprädikate

```
has_compage ("nikon", compage)
contains_productname (prodpage, "f90x")
has_productpage ("nikon", prodpage)
```

Wenn Sie sich die Operatorsignaturen angucken, dann wird klar, daß wir mit einem der `FindProdPage`–Operatoren weitermachen können, der nach Instantiierung und Ausführung vielleicht so aussehen könnte:

`FindProdPageOperator`

V: `in_trade ("nikon", "fotografie")`

```
    deals_in ("nikon", "f90x")
    relates_to ("fotografen", "fotografie")
N:  + has_productpage ("nikon", + "www.nikon.de/Produkte/")
    + contains_productname ("www.nikon.de/Produkte/", "f90x")
```

Damit sind zwei der drei Zielprädikate schon erfüllt! Für das letzte zu erfüllende Zielprädikat fällt die Wahl auch nicht weiter schwer:

`GuessCompageFromProdpageOperator`

```
V:  trade ("fotografie")
    has_productpage ("nikon", "www.nikon.de/Produkte/")
N:  + has_compage ("nikon", + "www.nikon.de")
    + in_trade ("nikon", "fotografie")
```

Daß hier das `in_trade`–Prädikat noch einmal erzeugt wird, ist nicht schlimm, da es schließlich immer noch Gültigkeit besitzt.

Durch Anwendung dieser beiden Operatoren haben wir alle Zielprädikate vollständig instantiieren können und damit das Gesamtziel erreicht. Natürlich sind noch vielfältige andere Kombinationsmöglichkeiten denkbar, die häufig auch durchgespielt werden, da die Operatoren bei der Ausführung ja auch scheitern können.

Vielleicht haben Sie ja jetzt Lust bekommen, ein wenig mit den Operatoren herumzuexperimentieren und sie sogar zu verbessern. Wir wünschen Ihnen auf jeden Fall viel Spaß dabei!

Literaturverzeichnis

[Anderson et al., 1998] Anderson, C. R., Smith, D. E., and Weld, D. S. (1998). Conditional effects in graphplan. In *Proceedings of AIPS-98.*

[Bartlet, 1961] Bartlet, F. (1932, revised 1961). *Remembering: A Study in Experimental and Spcial Psychology.* The University Press, Cambridge, UK.

[Beetz and McDermott, 1994] Beetz, M. and McDermott, D. (1994). Improving robot plans during their execution. In Hammond, editor, *Procs. of the Second International Conference on AI Planning Systems.* Morgan Kaufmann.

[Blum and Furst, 1997] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

[Bollaker et al., 1998] Bollaker, K. D., Lawrence, S., and Giles, C. L. (1998). *Citeseer: An Autonomous Web Agent for Automatic Retrieval and Identification of Interesting Publications.* NEC Research Institute, Princeton, NJ.

[Brachman, 1977] Brachman, R. (1977). What's in a concept: Structural foundations for semantic networks. *Int. Journal of Man-Machine Studies*, 9:127–152.

[Brachman and Levesque, 1985] Brachman, R. J. and Levesque, H. J., editors (1985). *Readings in Knowledge Representation.* Morgan Kaufmann.

[Brachman and Schmolze, 1985] Brachman, R. J. and Schmolze, J. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216.

[Braitenberg, 1984] Braitenberg, V. (1984). *Vehicles, Experiments in Synthetic Psychology.* MIT Press, Cambridge MA.

[CAPlan, 1998] CAPlan (1998). Caplan - computer assisted planning.
http://wwwagr.informatik.uni-kl.de/∼caplan/.

[Dumais et al., 1990] Dumais, Susan, T., Furnas, George, W., and Landauer, Thomas, K. (1990). Indexing by Latent Semantic Analysis. *Journal of the american society for information science*, 41(6):391–407.

[Etzioni et al., 1997] Etzioni, O., Golden, K., Weld, D., et al. (1997). Sound and efficient closed–world reasoning for planning. *Artificial Intelligence*, 82(1-2):113 – 148.

[Etzioni et al., 1992] Etzioni, O., Hanks, S., Weld, D., et al. (1992). An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115 – 125.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI.*

[Giles et al., 1998] Giles, L., Bollacker, K., and Lawrence, S. (1998). Citeseer: An automatic citation indexing system. In *ACM Conference on Digital Libraries*.

[Goodwin, 1993] Goodwin, R. (1993). Formalizing properties of agents. Technical Report CMU-CS-93-159, Carnegie Mellon University: School of Computer Science.

[Görz, 1995] Görz, G., editor (1995). *Einführung in die künstliche Intelligenz*. Addison-Wesley, 2 edition.

[Harrison, 1971] Harrison, Malcolm, C. (1971). Implementation of the Substring Test by Hashing. *Communications of the ACM*, 14(12):777–779.

[Hayes, 1985] Hayes, P. (1985). In defence of logic. In *Proc. of IJCAI-85*, pages 559–565.

[Hüllen et al., 1998] Hüllen, J., Bergmann, R., and Weberskirch, F. (1998). Webplan: Dynamix planning for domain-specific search in the internet. Technical report, University of Kaiserslautern, Dept. of ComputerScience, P.O. Box2049, D-67653 Kaiserslautern, Germany.

[Hunt et al., 1966] Hunt, E. B., Marin, J., and Stone, P. (1966). *Experiments in Induction*. Academic Press, New York.

[Javasoft, 2000] Javasoft (2000). What is the java (tm) platform?
http://www.javasoft.com/nav/whatis/.

[Joachims, 1997] Joachims, T. (1997). Text categorization with support vector machines: Learning with many relevant features. Technical Report 23, Universität Dortmund, LS VIII-Report.

[jwebserver, 2000] jwebserver (2000). Java (tm) web server.
http://www.sun.com/software/jwebserver.

[Klingspor, 1998] Klingspor, V. (1998). *Reaktives Planen mit gelernten Begriffen*. PhD thesis, Univ. Dortmund.

[Koehler et al., 1997] Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an adl subset. In *Proceedings of 4th European Conference on Planning*.

[Lawrence and Giles, 1998a] Lawrence, S. and Giles, L. (1998a). Inquirus, the neci meta search engine. In *International WWW Conference*, pages 95–105.

[Lawrence and Giles, 1998b] Lawrence, S. and Giles, L. (1998b). Searching the world wide web. *Science*, 280:98–100.

[Levesque, 1996] Levesque, H. J. (1996). What is planning in the presence of sensing? In *Proceedings of the 13th AAAI*. 1139–1146.

[Maida, 1987] Maida, A. S. (1987). *Encyclopedia of Artificial Intelligence*, volume 1, chapter Frame Theory, pages 302–312. John Wiley Sons, New York.

[McCallum et al., 1999] McCallum, A., Nigam, K., Rennie, J., and Seymore, K. (1999). Building domain-specific search engines with machine learning techniques. In *AAAI-99 Spring Symposium*.

[Minsky, 1974] Minsky, M. (1974). A framework for representing knowledge. Technical Report Artificial Intelligence Memo 306, MIT AI Lab.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw Hill.

[Morik, 1997] Morik, K. (1997). Einführung in die Künstliche Intelligenz. Skript zur gleichnamigen Vorlesung, 5.Auflage.

[Nebel, 1987] Nebel, B. (1987). *Reasoning and Revision in Hybrid Representation Systems.* Number 422 in Lecture Notes in AI. Springer, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Machine Learning. Morgan Kaufmann, San Mateo, CA.

[Rijsbergen, 1971] Rijsbergen, C., J. (1971). An algorithm for information structuring and retrieval. *The Computer Journal*, 14(4):407–412.

[Rosch, 1975] Rosch, E. (1975). Cognitive representations of semantic categories. *Journal of Experimental Psychology: General*, 104:192–233.

[Salton and Wong, 1978] Salton, G. and Wong, A. (1978). Generation and Search of Clustered Files. *ACM Transactions on Database Systems*, 3(4):321–346.

[Sander Beuermann, 1998] Sander Beuermann, W. (1998). Schatzsuche – die internet-suchmaschinen der zukunft. *c't*, 13.

[Selberg and Etzioni, 1995] Selberg, E. and Etzioni, O. (1995). Multi-service search and comparison using the metacrawler. In *Proceedings of the 1995 WWW Conference.*

[Selberg and Etzioni, 1997] Selberg, E. and Etzioni, O. (1997). The metacrawler architecture for resource aggregation on the web. *IEEE Expert.*

[Sheng–Te, 2000] Sheng–Te (2000). Java (tm) network ftp library. http://java-netlib.netpedia.net/.

[Smith and Weld, 1998] Smith, D. and Weld, D. (1998). Conformant graphplan. In *Proceedings of 15th Nat. Conf. AI.*

[Stevesoft, 1998] Stevesoft (1998). Regular expressions in java. http://www.javaregex.com.

[Sun, 2000] Sun (2000). Java (tm) 2 platform, standard edition product family. http://java.sun.com/jdk.

[Susan Gauch, 1996] Susan Gauch, Guijun Wang, M. G. (1996). Profusion*: Intelligent fusion from multiple, distributed search engines. Technical report, Department of Electrical Engineering and Computer Science (The University of Kansas), Cambridge (Mass.).

[Sutton, 1988] Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

[van Eylen, 1997] van Eylen, D. (1997). Altavista ranking of query results.

[Vapnik, 1982] Vapnik, V. (1982). *Estimation of Dependencies Based on Empirical Data.* Springer.

[Vapnik, 1998] Vapnik, V. (1998). *Statistical Learning Theory.* Wiley.

[Watkins, 1989] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

[Weld et al., 1998] Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty & sensing actions. *AAAI-98.*

[Wooldridge and Jennings, 1995] Wooldridge, M. J. and Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152.