Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-93-15

Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers

Robert Laux

Oktober 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers

Robert Laux

DFKI-D-93-15

Diese Arbeit wurde von Prof. Michael M. Richter und Dipl.-Inform. Christoph Klauck betreut.

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-9304/3 C4).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1993

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers

Robert Laux

November 1993

Inhaltsverzeichnis

Kapitel 1

Einleitung

Grammatiken sind allgemein als Beschreibungsmittel für formale Sprachen bekannt. Neben den klassischen Stringgrammatiken gibt es eine weitere Klasse von Grammatiken, die sogenannten *Graphgrammatiken*. Bei diesen bestehen die Produktionen nicht aus Zeichenketten, sondern aus Graphen.

Graphgrammatiken unterscheiden sich von Stringgrammatiken wesentlich durch ihre Komplexität. Bereits das Problem der Graph-Isomorphie, welches ein Teilproblem des eigentlichen Wortproblems darstellt, ist für allgemeine Graphgrammatiken als NP-Problem bekannt. Dies ist letztlich darin begründet, daß die Anzahl der Subgraphen eines Graphen im allgemeinen *exponentiell* mit dessen Größe wächst.

Parser für allgemeine Graphgrammatiken weisen somit ebenfalls eine Komplexität auf, die in dieser Komplexitätsklasse liegt. Diese Tatsache motiviert den Einsatz von domänenspezifischen *Heuristiken*, um trotzdem praktikable Laufzeiten zu erreichen. Eine Anwendung von Graphgrammatiken findet sich in der sogenannten *Feature-Erkennung*.

Feature-Erkennung und Graph-Parsing

Im Rahmen des Computer Integrated Manufacturing (CIM) wird die Automatisierung bzw. Computerunterstützung von Herstellungsprozessen vor allem in der Domäne des Maschinenbaus verfolgt. Am Herstellungsprozeß sind verschiedene Arbeitsbereiche (CAD, CAM, ...) beteiligt. Jeder dieser Bereiche arbeitet mit seinen eigenen Beschreibungsmitteln, den sogenannten *Featuren*.

Ein solches Feature ist ein bereichsspezifisches, qualitatives Beschreibungselement, das auf der geometrisch-technologischen Repräsentation aufsetzt und mit welchem ein Experte gewisse Informationen verbindet [2]. Mit einer Menge von Featuren, der *Feature-Sprache*, läßt sich das Produkt aus

der subjektiven Sicht eines Experten beschreiben.

Aufgabe der Feature-Erkennung ist es, die allgemein zugängliche geometrisch-technologische Produktbeschreibung in die Feature-Sprache eines bestimmten Arbeitsbereiches zu übersetzen. Somit kann die Feature-Erkennung als *Schnittstelle* zwischen Arbeitsbereichen, die unterschiedliche Sichten auf das Produkt haben bzw. eine andere Terminologie verwenden, angesehen werden. Ergebnis der Feature-Erkennung ist eine oder mehrere alternative sogenannte *Feature-Strukturen* des Produkts. Abbildung 1.1 zeigt eine Feature-Struktur eines Werkstücks.

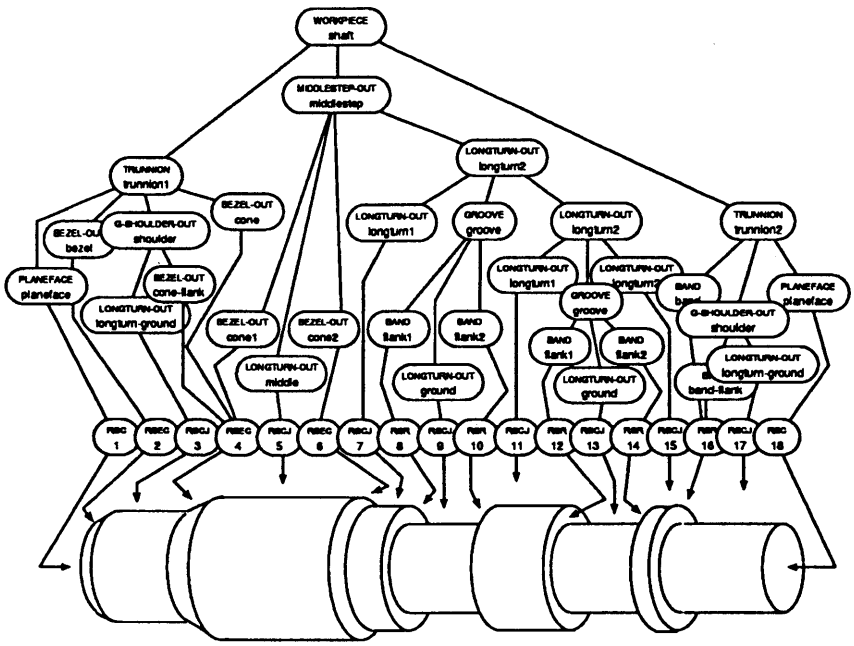


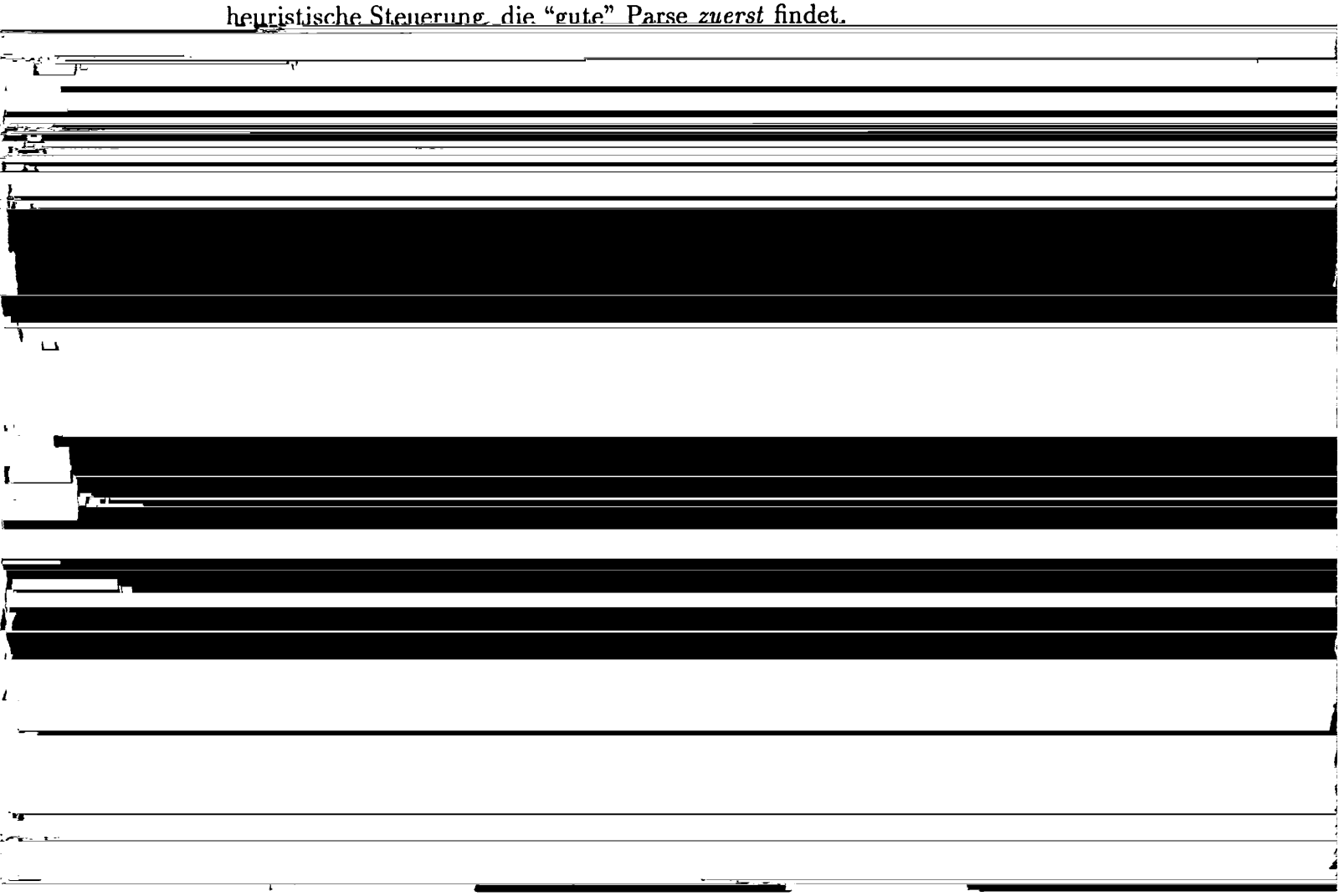
Abbildung 1.1: Feature-Struktur eines Werkstücks

Ein Ansatz zur Feature-Erkennung basiert auf Graphgrammatiken. Dabei werden Werkstücke durch Graphen und die Feature-Beschreibungen durch Produktionen einer Graphgrammatik (Featuregrammatik) beschrieben. Die terminalen Knoten eines Graphen repräsentieren elementare Flächenstücke, die Kanten stehen für topologische Relationen (Nachbarschaft, Überlappung) zwischen den Knoten. Somit stellt sich die Feature-Erkennung als Graph-Parsing Problem dar. Die vom Parser gefundenen Parsbäume sind gleichzeitig die Feature-Strukturen des Werkstücks.

Aufgabenstellung

Im Rahmen des Projekts ARC-TEC (Akquisition, Repräsentation und Compilation von TEChnischem Wissen) am Deutschen Forschungszentrum für Künstliche Intelligenz GmbH (DFKI) wurde der Grammatikformalismus 1-NRCFGG (NeighboRhood Controlled node labeled and node attributed Feature Graph Grammar) und der darauf zugeschnittene Graph-Parser GraPaKL entwickelt und auf das Problem der Feature-Erkennung von Werkstücken angewendet [12]. Aufgrund der exponentiellen Komplexität von GraPaKL soll in dieser Arbeit eine *Heuristikkomponente* zu dessen Steuerung entwickelt werden.

GraPaKL liefert zu einem als Graph gegebenen Werkstück *alle* möglichen Parse (Feature-Strukturen). Ist die Grammatik ambiguent, so existieren *mehrere* verschiedene Parse. Jedoch sind viele Parse nicht zufriedenstellend; man ist lediglich an "guten" (brauchbaren) Parsen[1] interessiert. Ziel ist also eine heuristische Steuerung, die "gute" Parse *zuerst* findet.



heuristischen Wissens soll ein Lernverfahren eingesetzt werden, da ein Experte auf einer anderen, höheren Ebene als der eines Parsing-Prozesses denkt und man somit keine Vorstellung davon hat, welche Entscheidung in einer bestimmten Situation während des Parsens zu treffen ist.

Der Begriff 'heuristisches Wissen' ist sehr weit gefaßt; im folgenden wird die Bezeichnung *Kontrollwissen* (Metawissen) für das zu akquirierende Wissen zur Steuerung von GraPaKL verwendet werden, da es auf einer höheren Ebene liegt und die Anwendung der Produktionen der Grammatik (Domänenwissen) explizit steuert.

Kapitel 2

GraPaKL

Dieses Kapitel gibt einen kurzen Überblick über GraPaKL, für den in dieser Arbeit eine Heuristikkomponente entwickelt wird. GraPaKL ist ein bottom-up arbeitender, agenda-basierter Chart-Parser, der für 1-NRCFGG'en entwickelt wurde.

Der Parsing-Algorithmus und die zugehörigen Datenstrukturen sollen hier nur insoweit dargelegt werden, wie es das Verständnis dieser Arbeit erfordert. Für eine detailliertere Betrachtung des Parsers sei hier auf [17] verwiesen.

2.1 Datenstrukturen

2.1.1 Patche

Bei 1-NRCFGG'en wird eine Produktion mit einem Knoten, ihrer linken Seite, assoziiert (siehe Abbildung 2.1). Die rechte Seite besteht aus einer Menge von *Rollen*, nämlich den Rollen des assoziierten Knotens, die mit geeigneten Knoten zu besetzen sind. Zwischen den Rollen bestehen funktionale und topologische Beziehungen (z.B. Konvexität, Nachbarschaft, Überlappung, usw.).

Eine Produktion ist eine Regel zur Ersetzung eines (Sub-)Graphen durch einen Knoten, wobei die rechte Seite der Produktion die Klasse der Graphen spezifiziert, auf die diese Regel anwendbar ist.

Der mit einer Produktion assoziierte Knoten besitzt ein Label (Sorte[1] der Produktion), verschiedene Attribute (geometrische Eigenschaften wie Bezugspunkt, Flächeninhalt, usw.) und Rollen.

Während des Parsens entstehen Instantiierungen der Produktionen durch

[1] Zur Definition einer Featuregrammatik gehört eine Menge von Sorten zusammen mit einer Partialordnung, der Sortenhierarchie.

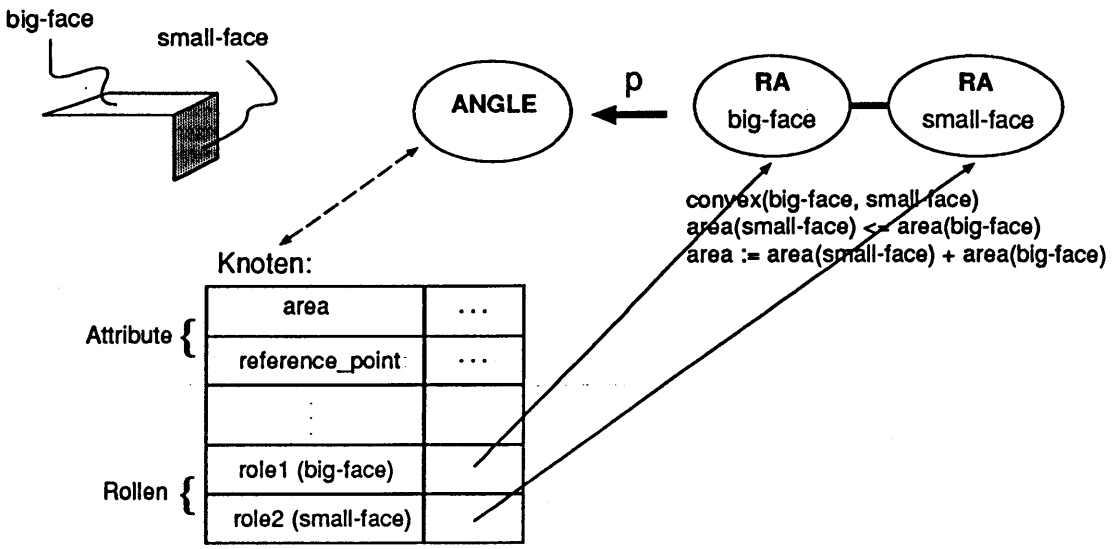


Abbildung 2.1: Eine NRCFGG-Produktion p der Sorte ANGLE und der damit assoziierte Knoten

Belegung ihrer Rollen mit Knoten, die die jeweiligen Rollenspezifikationen erfüllen.[2] Der Parslauf führt über teilweise instantiierte Produktionen (Hypothesen) hin zu vollständig instantiierten Produktionen (erkannte Features).

Die Instanz einer Produktion bzw. der damit assoziierte Knoten wird durch die Datenstruktur *Patch*[3] beschrieben. Ein Patch besteht im wesentlichen aus den folgenden Komponenten:

- Indentifikation der Produktion, deren Instanz das Patch ist
- Sorte des Patches (= Sorte der Produktion)
- Attribute des mit der Produktion assoziierten Knotens
- Rollenspezifikation der nächsten zu besetztenden Rolle

Ein Patch, bei dem alle Rollen belegt sind, heißt *komplett* (*cp*)[4], andernfalls *partiell* (*pp*)[5]. Abbildung 2.2 veranschaulicht ein partielles Patch der Sorte

SORT mit drei besetzten Rollen. Als nächstes wird nach einem geeigneten

[2] Die Reihenfolge, in der nach Rollenwerten für die Rollen gesucht wird, wird innerhalb der Spezifikation der Produktionen der Featuregrammatik mittels einer totalen Ordnung vom Grammatikentwickler (Experte) festgelegt (vgl. Abschnitt 2.3).

[3] *Patch* heißt in diesem Zusammenhang soviel wie (Teil-)Struktur.

[4] complete patch.

[5] partial patch.

Rollenwert für die (gemäß der innerhalb der Produktion spezifizierten Ordnung der Rollen) vierte Rolle gesucht. `ROLE-SPEC` bezeichnet die zugehörige Rollenspezifikation.

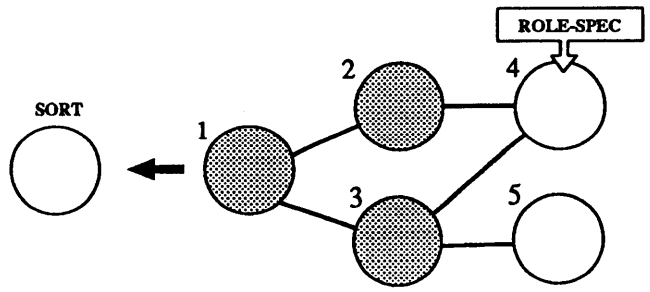


Abbildung 2.2: Ein partielles Patch mit drei besetzten Rollen

Eine Neuinstantiierung einer Produktion geschieht durch die Besetzung der *ersten* Rolle der Produktion mit einem bereits gefundenen *kompletten* Patch erzeugt, dessen erste Rolle durch *cp* belegt ist.

Eine weitere Vervollständigung eines partiellen Patches *pp* erfolgt durch Kombination mit einem bereits gefundenen *kompletten* Patch *cp*, das die Rollenspezifikation der *nächsten* zu besetzenden Rolle von *pp* erfüllt. In diesem Fall wird ein *neues* Patch erzeugt, indem eine *Kopie* *pp'* von *pp* angefertigt

Ein Vorteil der agenda-basierten Architektur besteht darin, daß man eine *flexible* Kontrollstruktur erhält. Es lassen sich durch die Reihenfolge der Auswahl der Patche aus der Agenda verschiedene Suchstrategien realisieren (siehe Abschnitt 2.2.2).

2.2 Der Parsing-Algorithmus

2.2.1 `Initialize`, `Choose` und `Combine`

Der Parsing-Algorithmus basiert im wesentlichen auf drei Operatoren, nämlich `Initialize`, `Choose` und `Combine`, die auf der Agenda und dem Chart arbeiten (siehe Abbildung 2.3).

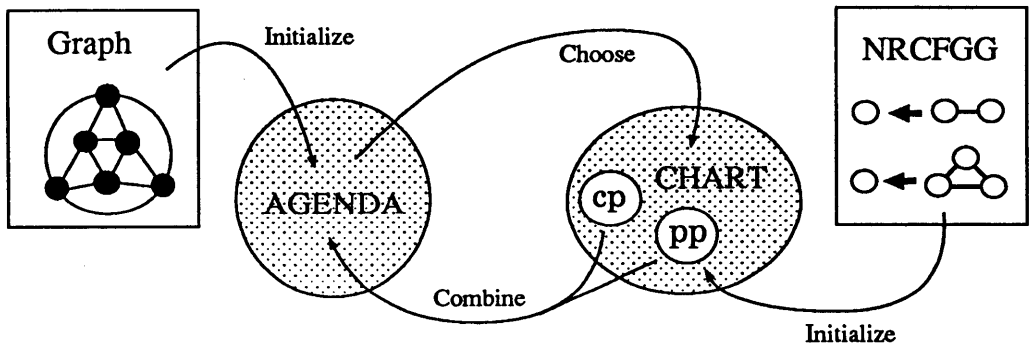


Abbildung 2.3: Die Arbeitsweise von GraPaKL

`Initialize` initialisiert den Chart für jede Produktion der Grammatik mit je einem partiellen Patch, bei dem *keine* Rolle belegt ist, und die Agenda mit je einem kompletten Patch für jeden (terminalen) Knoten des zu parsenden Graphen.

In jedem Schritt wählt `Choose` ein Patch aus der Agenda aus und `Combine` kombiniert das Patch mit den passenden Patchen aus dem Chart. Die durch `Combine` neu entstandenen Patche werden in die Agenda eingetragen, das aus der Agenda ausgewählte "Urpatch" wandert in den Chart und wird aus der Agenda gelöscht.[7]

`Choose` und `Combine` werden solange abwechselnd angewendet, bis die

[7]Die durch `Combine` entstandenen *Parse* werden nicht in die Agenda bzw. den Chart eingetragen, sondern in einer separaten Datenstruktur gesammelt.

Agenda schließlich *leer* ist.[8] Abbildung 2.4 zeigt nocheinmal den GraPaKL-Basisalgorithmus.

```
begin
    Initialize agenda and chart;
    repeat
        Choose patch from agenda;
        Combine patch;
        add new patches to agenda;
        delete patch in agenda;
        add patch to chart;
    until agenda = ∅;
end;
```

Abbildung 2.4: Der GraPaKL Basis-Algorithmus

2.2.2 Suchstrategien

Der Choose-Operator ist *nichtdeterministisch* in der Hinsicht, daß keine Aussage darüber gemacht ist, welches Patch aus der Agenda auszuwählen ist, falls mehrere Patche in der Agenda vorhanden sind (was fast immer der Fall ist). Dies entspricht der in Abschnitt 2.1.2 angesprochenen flexiblen Kontrollstruktur des Parsers.

Beispielsweise erhält man Breitensuche, falls immer das am längsten in der Agenda befindliche Patch ausgewählt wird und Tiefensuche, falls immer das zuletzt in die Agenda gekommene Patch ausgewählt wird.

Heuristische Suchstrategien lassen sich durch die Verwendung von Kontrollwissen realisieren, welches die Auswahl der Patche aus der Agenda und somit die Suche des Parsers steuert.

2.2.3 Terminierung, Fairneß und Vollständigkeit

Die Terminierung des Parsers kann für allgemeine Grammatiken nicht sichergestellt werden. Lediglich für solche Grammatiken, die die sogenannte Kettenbedingung und die Bedingung der lokalen Endlichkeit erfüllen, d.h.

[8] Bei der zu entwickelnden heuristischen Steuerung des Parsers wird die Abbruchbedingung agenda = ∅ ersetzt durch (agenda = ∅ ∨ parslist ≠ ∅), dā "gute" Parse zuerst gefunden werden sollen. Dabei bezeichnet parslist die Menge der gefundenen Parse.

deren Ableitungsbaum endlich ist, kann die Terminierung garantiert werden [12].

Verbunden damit ist das Problem der *Fairneß*[9]. Terminiert der Parser für eine Grammatik, so ist natürlich auch die Fairneß gegeben, da die Agenda schließlich leer ist, also alle erzeugbaren Patche aus der Agenda ausgewählt wurden. Für Grammatiken, bei denen der Parser eventuell nicht terminiert, hängt die Fairneß von der Suchstrategie ab. Während Breitensuche die Fairneßbedingung erfüllt, muß dies bei Tiefensuche und heuristischen Suchstrategien nicht mehr der Fall sein, d.h. daß Patche eventuell nie aus der Agenda ausgewählt werden. Eine heuristische Steuerung sollte jedoch nur solche Patche außer Acht lassen, die zum Finden eines "guten" Parses *nicht* beitragen können.

Die Erfahrung hat gezeigt, daß *sinnvolle* Featuregrammatiken die beiden oben genannten Bedingungen erfüllen, so daß angenommen werden kann, daß der Parser für diese Art von Grammatiken *immer* terminiert. Der Chart enthält dann *alle* Parse und *alle* Patche, die sich hinsichtlich der, Featuregrammatik und des zu parsenden Graphen erzeugen lassen. Der Parser ist also im Falle der Terminierung *vollständig* [12].

2.3 Ansatzpunkte für eine heuristische Steuerung

Die Arbeitsweise des Parsers bietet zwei prinzipielle Möglichkeiten einer heuristischen Steuerung.

Die erste Möglichkeit besteht in einer (totalen) Ordnung der Rollenspezifikationen für jede Produktion der Grammatik. Diese Ordnung legt fest, in welcher Reihenfolge die Rollen einer Produktion zu besetzen sind. Die Reihenfolge sollte so gewählt werden, daß der Parser diejenigen Rollen, deren Rollenspezifikation "schwierig" zu erfüllen sind, möglichst früh zu besetzen versucht.[10] Folglich werden die "leichter" zu besetzenden Rollen erst dann betrachtet, wenn für die anderen schon Rollenwerte gefunden wurden und somit die Wahrscheinlichkeit, daß das Patch vervollständigt werden kann, relativ hoch ist. Dem Parser bleibt also "unnötige" Arbeit erspart.

Die zweite Möglichkeit besteht in einer Ordnung der in der Agenda befindlichen Patche (vgl. Abschnitt 2.2.2), so daß immer das vielversprechend-

[9]Der Parser arbeitet *fair*, falls jedes erzeugte Patch auf jeden Fall irgendwann aus der Agenda ausgewählt wird.

[10]Die "Schwierigkeit" der Erfüllbarkeit einer Rollenspezifikation könnte beispielsweise an der Sorte der Rolle (selten vorkommende Sorte) oder an der Anzahl der mit der Rollenspezifikation verbundenen Constraints gemessen werden.

ste Patch im Hinblick auf das Finden eines "guten" Parse ausgewählt wird.[11] Während erstere bereits in GraPaKL eingebaut ist, ist die heuristische Steuerung über die Ordnung der Patche in der Agenda Gegenstand dieser Arbeit.

Beide Arten der heuristischen Steuerung erfordern Kontrollwissen. Bei der Ordnung der Rollenspezifikationen ist dieses Wissen *explizit*; der Benutzer (Experte) legt bei der Definition der Grammatik die Ordnung fest. Somit obliegt diese heuristische Steuerung *vollständig* dem Benutzer. Für die Ordnung der Patche in der Agenda werden im folgenden Kapitel verschiedene Arten von Kontrollwissen diskutiert.

[11] Man beachte, daß dafür *keine* Änderung der GraPaKL-Kontrollstruktur vorgenommen werden muß (vgl. Abschnitt 2.2.2).

Kapitel 3

Lernen von Kontrollwissen

Zur heuristischen Steuerung von GraPaKL über die Ordnung der Patche in der Agenda wird *Kontrollwissen* benötigt, welches die Auswahl der Patche aus der Agenda steuert. In der Einleitung wurde deutlich gemacht, daß es notwendig ist, dieses Wissen mit Hilfe eines Lernverfahrens zu akquirieren.

In diesem Kapitel werden zunächst verschiedene Arten von Kontrollwissen diskutiert. Anschließend werden Lernverfahren betrachtet, die sich zur Akquisition dieses Wissens eignen.

3.1 Arten von Kontrollwissen

3.1.1 Bewertungsfunktionen

Eine Art von Kontrollwissen sind *Bewertungsfunktionen* $b : \mathcal{A} \mapsto \mathcal{V}$ ($\mathcal{A}$ bezeichnet die Menge der Alternativen und $\mathcal{V}$ eine (eventuell partiell) geordnete Menge) zur Berechnung von *Prioritäten* für verschiedene Alternativen.

Es werden sowohl lineare Funktionen der Form

$$b(x) = a_1 \cdot f_1(x) + a_2 \cdot f_2(x) + \ldots + a_n \cdot f_n(x)$$

wobei die Koeffizienten $a_i \in \mathbb{R}$ und die $f_i : \mathcal{A} \mapsto \mathcal{W}_i$ ($\mathcal{W}_i$ bezeichnet den Wertebereich von f_i) die Funktionen zur Berechnung der zur Bewertung herangezogenen Kriterien[1] sind (z.B. Anzahl der Bauern beim Schach, Anzahl von Literalen in einer Klausel bei Theorembeweisern, usw.) als auch nichtlineare Funktionen verwendet. Alle verwendeten (numerischen und symbolischen) Kriterien müssen also auf *numerische* Werte mittels der f_i abgebildet werden.

[1] In der Literatur wird anstatt 'Kriterium' oft der Begriff *Feature* (*feature-basierte* Bewertung) verwendet. Um jedoch Kollisionen mit dem Feature-Begriff aus der Werkstückbeschreibung zu vermeiden, soll im folgenden der Begriff 'Kriterium' verwendet werden.

Diese Art von Kontrollwissen ist *implizit* und hat eher *subsymbolischen* Charakter, da es ist in die Parameter der Funktionen hineinkodiert ist. Im Gegensatz zu symbolischem Wissen sieht man eigentlich nicht, *was* gelernt wurde. Es stellt sich aber die Frage, ob dies für die Anwendung erforderlich ist.

Bewertungsfunktionen werden sehr oft bei Spielen (z.B. [26], [16]) verwendet. Andere Beispielanwendungen sind Theorembeweiser (z.B. [3], [28]) und Scheduling-Probleme (z.B. [11]).

3.1.2 Kontrollregeln

Eine weitere, häufig verwendete Art von Kontrollwissen sind *Kontroll-* bzw. *Metaregeln* [5]. Durch die Interpretation dieser Produktionsregeln werden Alternativen (Zustände oder Operatoren) ausgeschlossen oder zur Fortsetzung empfohlen. Die Syntax für eine Kontrollregel könnte beispielsweise folgendermaßen aussehen:

$$\textbf{if } C_1 \wedge \ldots \wedge C_n \textbf{ then } \left\{ \begin{array}{c} sort \\ production \end{array} \right\} will \left\{ \begin{array}{c} probably \\ definitely \end{array} \right\} be \left\{ \begin{array}{c} useful \\ useless \end{array} \right\}$$

wobei die C_i die zu erfüllenden Vorbedingungen sind.

Diese Art von Wissen ist *symbolisch* und liegt *explizit* in Form von Produktionsregeln in einer Wissensbasis vor. Der Vorteil gegenüber einer Bewertungsfunktion besteht darin, daß man hier sieht, *was* gelernt wurde. Jedoch wird hier zusätzlich ein Regelinterpreter benötigt, weshalb Regeln sich besonders für Systeme eignen, deren Domänenwissen bereits in Form von Produktionsregeln vorliegt.

Wird die Anzahl der Kontrollregeln zu groß, so kann es passieren, daß der Aufwand zur Verarbeitung dieser die Ersparnis im Suchraum wieder egalisiert oder gar übertrifft, so daß sich insgesamt keine Verbesserung oder sogar eine Verschlechterung ergibt; dies ist das sogenannte *Utility Problem* [20]. Deshalb sind Produktionsregeln in Domänen, wo das heuristische Wissen sehr reichhaltig ist, wie beispielsweise bei Schach, ungeeignet. Bei der Interpretation von Regeln ist man meist mit einer *Konfliktmenge* konfrontiert, so daß zusätzlich eine Strategie zur Konfliktlösung erforderlich ist.

Systeme, die Kontrollregeln verwenden sind beispielsweise SOAR [13], PRODIGY [19], SAGE [14], LEX [21]. Bei all diesen Systemen handelt es sich um allgemeine, regelbasierte Problemlöser.

3.1.3 Präferenzprädikate

Präferenzprädikate $\mathcal{P} : \mathcal{A} \times \mathcal{A} \mapsto \{0, 1\}$, die bezüglich *zweier* Alternativen

bestimmen, ob die eine der anderen vorzuziehen ist, stellen eine weitere Art von Kontrollwissen dar. Durch paarweise Vergleiche lassen sich die vielversprechendsten Alternativen herausfiltern.

Ein Präferenzprädikat läßt sich sowohl als Entscheidungsbaum (explizit) als auch als mathematischer Ausdruck (implizit) realisieren. Gegenüber einer Bewertungsfunktion wird eine Alternative nicht relativ zu allen, sondern nur relativ zu *einer* anderen Alternative bewertet. Somit ist eine feinere Differenzierung der Alternativen möglich; bei Bewertungsfunktionen ist mit der Berechnung der Priorität einer Alternative deren Güte relativ zu den anderen Alternativen bereits vollkommen festgelegt.

Die Anzahl der benötigten paarweisen Vergleiche steigt jedoch sehr schnell mit der Anzahl der Alternativen, so daß sich ein hoher Berechnungsaufwand ergibt, falls in jedem Schritt "viele" Alternativen zur Verfügung stehen.

Ein Beispielsystem, das ein Präferenzprädikat in Form eines Entscheidungsbaumes verwendet ist PREFER [30], wo es um die Lösung des "8-Puzzle" geht.

3.1.4 Fälle

Im weiteren Sinne lassen sich auch *Fälle* als Kontrollwissen auffassen. Unter Fällen sollen dabei vollständige, aus der *Erfahrung* bekannte (d.h. mit einem konkreten Ereignis korrespondierenden) Problemlösungen verstanden werden, die in einer *Fallbasis* gehalten und zur Lösung von aktuellen Problemen herangezogen werden. Fälle werden somit auch als *episodisches* Wissen bezeichnet. Die Abgrenzung des Fallbegriffs gegen den Begriff der Regel besteht darin, daß (heuristische) Regeln durch einen Generalisierungsprozeß aus mehreren Ereignissen (Problemlösungen) hervorgehen [1], wobei die konkreten Problemlösungen danach *nicht* mehr aufbewahrt werden.

Ein Beispiel für eine *fallbasiert* gesteuerte Suche ist [4]. Dabei geht es um die Lösung des "8-Puzzle", wobei ein Fall eine Sequenz von Auswahlentscheidungen, die einen gewissen Zustand (Spielsituation) in einen Zielzustand überführt. Wegen der großen Anzahl der möglichen Zustände ist es zu aufwendig, für jeden Zustand einen Fall in der Fallbasis abzulegen. Deshalb werden die Zustände mittels einer Indexabbildung in verschiedene Indizes kodiert, die "Äquivalenzklassen" von Zuständen beschreiben.[2] Während der Suche wird dann mit Hilfe des Index des aktuellen Zustandes in der *Fallbasis* nach einem geeigneten Fall (Sequenz von Indizes) gesucht, mit Hilfe dessen mögliche Folgezustände ausgeschlossen bzw. weiterverfolgt werden können.

[2] Diese Indexabbildung ist vergleichbar mit dem beim fallbasierten Schließen verwendeten *Ähnlichkeitsmaß*.

Die Verwendung von Fällen erfordert die Verwaltung einer Fallbasis, die effiziente Such- und Zugriffsoperationen zur Verfügung stellt. Ein weiteres Problem besteht in einer günstigen Wahl der Indexabbildung; dies kann als Lerngegenstand angesehen werden. Darüberhinaus kann es vorkommen, daß zu einem gewissen Zustand keine geeigneten Fälle in der Fallbasis vorhanden sind, so daß kein Ausschluß von Alternativen möglich ist; dann muß eine "Default-Heuristik" zum Einsatz kommen, die das weitere Vorgehen bestimmt.

3.1.5 Diskussion

Zunächst einige Feststellungen, die zur Beurteilung der verschiedenen Arten von Kontrollwissen zur Steuerung von GraPaKL relevant sind:

- die Güte der Patche in der Agenda ändert sich dynamisch mit dem "Zustand" des Parsers,
- die Anzahl der Alternativen (Patche) in jedem Schritt ist *sehr groß*[3],
- man hat keine *genaue* Vorstellung davon, wie die Auswahlentscheidungen zu treffen sind, so daß "gute" Parse zuerst gefunden werden und
- das benötigte Kontrollwissen ist *unscharfer* Natur.

Unter dem Zustand des Parsers soll die Anzahl der bereits erkannten Features (komplette Patche) der verschiedenen Sorten verstanden werden. Dies wird in Abschnitt 4.2.3 formal definiert.

Um ein Gefühl dafür zu bekommen, daß das Kontrollwissen unscharfer Natur ist, kann man sich vorstellen, daß die Priorität der Patche einer Sorte *je* größer wird, *desto* mehr komplette Patche einer anderen Sorte bereits erkannt wurden oder daß es sinnvoll ist, Patche einer bestimmten Produktion dann zu bevorzugen, wenn *wenige/einige/viele* komplette Patche einer anderen Produktion bereits erkannt wurden. Es kommt also *nicht* auf die genaue Anzahl der Patche, sondern auf unscharfe Situationen an.

Für die Modellierung dieses unscharfen, "kontinuierlichen" Wissens eignet sich eine *subsymbolische* bzw. *numerische* Repräsentation. Ungeeignet sind jedoch Produktionsregeln, deren Vorbedingungen *entweder* erfüllt sind, so daß die Regel feuert *oder* nicht ("exact match").

Produktionsregeln bewähren sich eher in Domänen, wo (scharfes) heuristisches Wissen direkt formuliert oder durch Inferenzen auf vorhandenem

[3]Bei realistischen Featuregrammatiken ist es nicht ungewöhnlich, daß sich zu einem Zeitpunkt hundert oder mehr Patche in der Agenda befinden.

Wissen über die Domäne akquiriert werden kann. Dafür muß jedoch bekannt sein, wie man sich in einer gewissen Situation zu verhalten hat oder es muß (reichhaltiges) symbolisches Domänenwissen zur Verfügung stehen. Dies ist hier jedoch nicht der Fall.

Wegen der sehr großen Anzahl von Alternativen in jedem Schritt und der ständigen Änderung der Güte der Patche scheidet ein Präferenzprädikat zur Steuerung von GraPaKL aus (vgl. Abschnitt 3.1.3).

Die Verwendung von Fällen zur Steuerung der Suche scheint hier zu zu aufwendig. Beim Parsen komplexer Werkstücke sind eventuell mehrere hundert Schritte (= Auswahl eines Patches aus der Agenda) des Parsers notwendig, so daß eine Suche nach ähnlichen Situationen in der Fallbasis nach jedem Schritt nicht möglich ist.

Als Konsequenz wird für die Steuerung von GraPaKL eine Bewertungsfunktion verwendet. Die konkrete Bewertungsfunktion ist Gegenstand von

Abschnitt 4.2.3. Im nächsten Abschnitt sollen Lernverfahren vorgestellt werden, die sich zur Akquisition einer Bewertungsfunktion eignen.

3.2 Lernen von Bewertungsfunktionen

In diesem Abschnitt werden Lernverfahren vorgestellt, die sich zur Akquisition einer Bewertungsfunktion einsetzen lassen. Für die Beurteilung der Eignung der verschiedenen Verfahren sind folgende Aspekte von besonderer Relevanz:

- Können *"beliebige"* Funktionen oder nur Parameter einer vorgegebenen Funktion gelernt werden ?
- Ist das Verfahren *inkrementell* ?
- Werden die Trainingsdaten auf nicht angelernte Eingaben angemessen *verallgemeinert* ?
- Werden *irrelevante* Bewertungskriterien entdeckt und entsprechend gehandhabt ?
- Werden *Korrelationen* zwischen Bewertungskriterien berücksichtigt ?

Letzteres soll im folgenden erläutert werden. In den meisten Fällen sind die Bewertungskriterien untereinander nicht unabhängig. Beispielsweise wird in [9] eine Bewertung von Hornklauseln vorgenommen mit dem Ziel, daß das PROLOG-System einen Beweis möglichst früh findet. Dabei werden u.a. die beiden Bewertungskriterien *Anzahl der Literale* ($\#L$) und *Anzahl der negativen Literale* ($\#nL$) verwendet. Zwischen diesen besteht eine funktiona-

beim Lernen der Bewertung nicht berücksichtigt, so ergibt sich eine *verzerrte* Gewichtung der Kriterien relativ zueinander und somit eine schlechtere Suchleistung.

3.2.1 Rote Learning

Rote Learning [23] stellt eine rudimentäre Form von Lernen dar. Dabei werden Zustände bzw. Alternativen zusammen mit ihrer Bewertung gespeichert und bei Bedarf abgerufen. Dies eignet sich besonders dann, wenn die Bewertungen durch *Vorausschau*, wie etwa bei Spielen, während der Suche gewonnen werden (vgl. 3.2.2) und somit die Vorausschau durch das eventuell billigere "Abrufen" der Bewertung ersetzt werden kann.

In den meisten Anwendungen zu aufwendig, alle möglichen Zustände mit ihrer Bewertung zu speichern. Deshalb muß eine geeignete Verallgemeinerung der Zustände vorgenommen werden. Eine solche Verallgemeinerung kann selbst als Lernen einer Abbildung angesehen werden, so daß sich wieder das ursprüngliche Problem der Akquisition einer Funktion ergibt.

Rote Learning wird beispielsweise im Dame-Spielprogramm von Samuel verwendet [26].

Vorteil

+ Eventuell Einsparung von Berechnungsaufwand bei Suche mit (weiter) Vorausschau (z.B. Spiele).

Nachteil

- Eventuell aufwendige Suche nach der Bewertung für den aktuellen Zustand.

3.2.2 Temporal Difference Method

Bei der *Temporal Difference Method* [29] werden die Parameter einer Bewertungsfunktion *während* der Suche angepaßt. Dabei wird die Güte des Zustandes, in dem man sich gerade befindet durch *Vorausschau* (mit der aktuellen Bewertungsfunktion) ermittelt und mit dem tatsächlichen Wert der Bewertungsfunktion (also ohne Vorausschau) für diesen Zustand verglichen. Die Parameter sind so zu verändern, daß sich die Differenz zwischen beiden Werten verringert. Diese Methode eignet sich selbverständlich nur für solche Probleme, bei denen eine Vorausschau Sinn macht (wie z.B. bei Spielen).

Diese Methode wird neben Rote-Learning ebenfalls von Samuel in seinem Dame-Spielprogramm verwendet [26].

Vorteil

+ Das Verfahren ist inkrementell.

Nachteile

- Es werden die Parameter einer vorgegebenen Funktion verändert, d.h. eine "beliebige" Funktion kann nicht gelernt werden.
- Korrelationen zwischen Bewertungskriterien werden nicht berücksichtigt.

3.2.3 Lineare Regression

Eine weitere Methode zur Akquisition einer Bewertungsfunktion ist die *lineare Regression* (z.B. [6], [31]). Bei dieser aus der Statistik bekannten Methode werden die Parameter einer linearen Funktion so gewählt, daß die entstehende *Regressionsgerade* den Verlauf der Trainingsdaten (Ein-/Ausgabepaare der zu lernenden Bewertungsfunktion) so gut wie möglich approximiert. Dazu wird meistens das *Prinzip der kleinsten Quadrate* verwendet; dieses besagt, daß die Gerade so zu legen ist, daß die Summe der Quadrate aller Abstände der Trainingsdaten zu der Geraden möglichst klein wird (siehe Abb. 3.1).

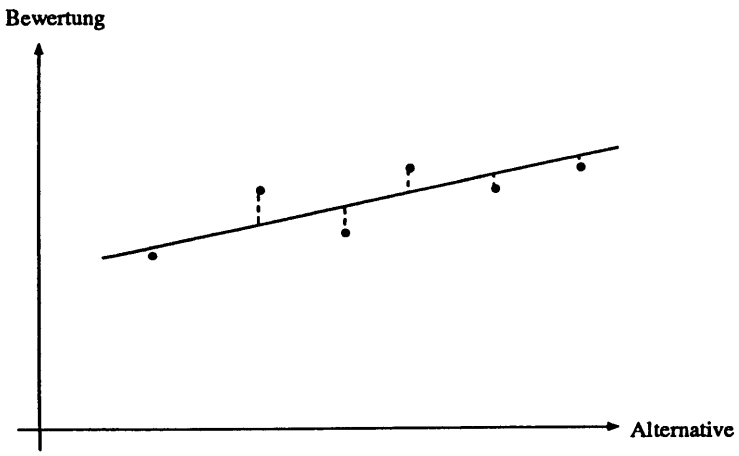


Abbildung 3.1: Lineare Regression

Eine gewisse Nichtlinearität kann dadurch erreicht werden, daß der "Kriterienraum" in verschiedene Bereiche aufgeteilt wird; in jedem Bereich wird dann eine *lokale* Bewertungsfunktion mittels Regression berechnet und man

erhält insgesamt eine stückweise lineare (globale) Bewertungsfunktion (siehe [22]).

Beispielsysteme, die lineare Regression verwenden, sind MULTIPLE [27] und PLS [22]. Bei MULTIPLE handelt es sich um einen Beweiser, während PLS ein allgemeines Suchprogramm (Problemlöser) ist.

Vorteil

+ Das Verfahren ist einfach in der Realisierung.

Nachteile

- Es werden die Parameter einer linearen Funktion verändert. Für eine gewissen Grad an Nichtlinearität ist zusätzlicher Aufwand nötig.
- Das Verfahren ist nicht inkrementell.

3.2.4 Genetisches Lernen

Beim *genetischen Lernen* ([10],[8]) einer Bewertungsfunktion werden ausgehend von einer *Anfangspopulation* (initiale Menge von Bewertungsfunktionen) schrittweise neue *Generationen* Populationen (*Generationen*) von Bewertungsfunktionen durch die Anwendung der genetischen Operatoren `Selektion`, `Rekombination` und `Mutation` erzeugt. Dabei ist eine Bewertungsfunktion b durch eine Zeichenkette, die eine Kodierung der Parameter von b ist, repräsentiert. Es werden also nicht die Parameter von b direkt, sondern deren Kodierungen verändert. Die Vorstellung dabei ist, daß die einzelnen Zeichen die *Gene* der Bewertungsfunktion darstellen.

Die Hoffnung ist, daß die "guten" *Individuen* einer Population (Bewertungsfunktionen) ihre Eigenschaften in neuen, besseren Individuen vereinigen und daß die "schlechten" *Individuen* aussterben. Abbildung 3.2 zeigt den genetischen Basisalgorithmus.

Beim Erzeugen einer neuen Generation wird zunächst für jedes Individuum einer Population dessen Güte, die sogenannte *Fitneß*, durch Testläufe bestimmt. Die `Selektion` nimmt eine Auslese der "fittesten" Individuen durch Zufallsexperimente vor; dabei wird die Fitneß eines Individuums als dessen Überlebenswahrscheinlichkeit aufgefaßt. Die paarweise `Rekombination` (Vermehrung) der durch die `Selektion` ausgelesenen Individuen geschieht durch die Vertauschung eines Teils deren Zeichenketten. Dies ist in Abbildung 3.3 verdeutlicht; die Zeichenketten der Elternindividuen werden ab dem Strich einfach vertauscht, so daß zwei neue Individuen entstehen, die je einen Teil der Gene beider Individuen in sich vereinigen. Durch die `Mutation` werden

```
begin
   wähle Anfangspopulation G_0;
   i ← 0;
   loop
      bestimme die Fitneß der Individuen aus G_i;
      if G_i enthält zufriedenstellendes Individuum
         then halt;
         else
            i ← i + 1;
            wähle die vermehrungswürdigen Individuen aus (Selek-
            tion);
            erzeuge neue Generation G_i durch Rekombination und
            Mutation;
   endloop;
end;
```

Abbildung 3.2: Prinzip des genetischen Lernens

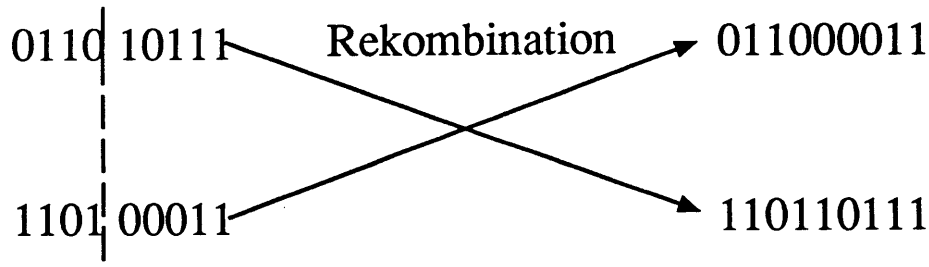


Abbildung 3.3: Rekombination zweier Individuen

zusätzlich Individuen (zufällig) verändert und in die neue Generation aufgenommen. Es werden solange neue Generationen erzeugt, bis genügend "gute" Individuen (Bewertungsfunktionen) gefunden wurden.

Der Hauptvorteil genetischen Lernens besteht darin, daß gleichzeitig in verschiedenen Regionen des Suchraumes gesucht wird, so daß die Wahrscheinlichkeit, dem globalen Optimum sehr nahe zu kommen (gegenüber "lokalen" Suchverfahren) relativ hoch ist.

Ein Beispiel für die Verwendung genetischen Lernens zur Akquisition einer Bewertungsfunktion ist [11]. Dort geht es um das "3D-Packproblem", also im Prinzip um ein Scheduling-Problem.

Vorteil

+ "Paralleles" Suchen an verschiedenen Stellen des Suchraumes.
+ Es werden keine Trainingsdaten benötigt

Nachteile

- Es werden die Parameter einer vorgegebenen Funktion verändert, d.h. eine "beliebige" Funktion kann nicht gelernt werden.
- Für die Ermittlung der Fitneß jedes Individuums sind aufwendige Testläufe erforderlich.
- Es muß eine geeignete Anfangspopulation erzeugt werden.

3.2.5 Backpropagation

Bei *Backpropagation* handelt es sich um ein konnektionistisches Lernverfahren. Dabei ist die Bewertungsfunktion als vollständig vernetztes Feedforwardnetz[4] (Backpropagationnetz) realisiert (Abb. 3.4), dessen Gewichte durch den *Backpropagationalgorithmus* [25] bezüglich einer Menge $\mathcal{T}$ von Trainingsinstanzen eingestellt werden.

Als Aktivierungsfunktion der Neuronen wird die Sigmoidfunktion

$$A(x) = \frac{1}{1 + e^{\frac{-x}{T}}}$$

verwendet, wobei T der *Temperaturparameter* ist. Die Aktivität a eines Neurons j ist dann gegeben durch $a_j = A(net_j)$, wobei

$$net_j = \sum_i c_{ij} \cdot a_i$$

[4]Bei einem *Feedforwardnetz* handelt es sich um ein geschichtetes neuronales Netz, bei dem nur Verbindungen zwischen Neuronen verschiedener Schichten in Richtung der Ausgabeschicht bestehen.

der *Nettoinput* am Neuron j ist und dabei über alle Neuronen i summiert wird, von denen eine Verbindung zu Neuron j besteht.

Die Trainingsmenge $\mathcal{T}$ besteht aus Trainingsinstanzen der Form (in, out), wobei *out* der gewünschte Ausgabevektor (Funktionswert) zum Eingabevektor *in* ist. Der Lernvorgang für eine Trainingsinstanz besteht aus zwei Phasen, der *Vorwärts-* und der *Rückwärtsphase*. In der Vorwärtsphase wird lediglich der Eingabevektor *in* durch das Netz propagiert, um die tatsächliche Netzausgabe zu berechnen. In der Rückwärtsphase wird die Differenz zwischen der tatsächlichen Ausgabe und der gewünschten Ausgabe *out* berechnet und rückwärts Schicht für Schicht durch das Netz propagiert, wobei die Gewichte gemäß der verallgemeinerten *Deltaregel* verändert werden (im folgenden bezeichnen i und j Neuronen, die in zwei benachbarten Schichten liegen, wobei eine Verbindung nur von Neuron i zu Neuron j mit der Verbindungsstärke c_{ij} besteht):

$$\Delta c_{ij} = \eta \cdot \delta_j \cdot a_i$$

wobei $\eta \in [0, 1]$ die *Lernrate*, a_i die Aktivität von Neuron i ist und δ_j folgendermaßen berechnet wird:

$$\delta_j = (out_j - a_j) \cdot A'(net_j)$$

falls j ein Ausgabeneuron ist und

$$\delta_j = (\sum_k \delta_k \cdot c_{jk}) \cdot A'(net_j)$$

falls j ein verborgenes Neuron ist.

Die Deltaregel realisiert nichts anderes als die *Methode des steilsten Abstiegs* in dem durch die Fehlerfunktion

$$E = \frac{1}{2} \sum_{(in,out)\in\mathcal{T}} \sum_{i\in O} (out_i - a_i)^2$$

gegebenen "Fehlergebirge", wobei O die Menge der Ausgabeneuronen und out_i die i-te Stelle des (gewünschten) Ausgabevektors *out* ist. Ziel ist es also, ein (möglichst das globale) Minimum der Fehlerfunktion E zu erreichen.

Damit sich das Netz schneller gegen die gewünschte Abbildung "konvergiert", kann zusätzlich ein sogenannter *Momentum-Term* in die Berechnung der Gewichtsänderungen aufgenommen werden:

$$\Delta c_{ij} = \eta \cdot \delta_j \cdot a_i + \alpha \cdot \Delta c^*_{ij}$$

wobei $\alpha \in [0, 1]$ und Δc^*_{ij} die letzte Gewichtsänderung ist. Somit wird verhindert, daß um ein Minimum der Fehlerfunktion herumgependelt, dieses aber nicht erreicht wird.

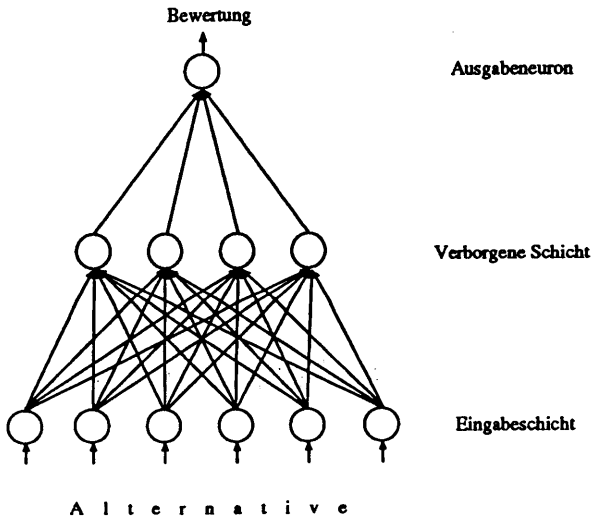


Abbildung 3.4: Bewertungsfunktion als neuronales Netz

Durch die Verwendung verborgener Schichten lassen sich "beliebige" Funktionen realisieren.

Das Problem bei Backpropagation besteht darin, daß die Minimierung der Fehlerfunktion eventuell in einem *lokalen* Minimum hängen bleibt. Jedoch läßt sich durch geeignete Wahl der Parameter bzw. durch wiederholtes Lernen mit anderen Anfangsgewichten ein Minimum erreichen, daß dem globalen Minimum sehr nahe kommt.

Systeme, bei denen Backpropagation zur Akquisition von Kontrollwissen eingesetzt wird, sind SETHEO [28] und CRECK [9]. Ersteres ist ein Theorembeweiser, letzteres eine Steuerungskomponente, die ein PROLOG-System bei der Beweissuche unterstützt.

Vorteile

+ Es lassen sich "beliebige" Funktionen realisieren.
+ Korrelationen zwischen Bewertungskriterien werden berücksichtigt.
+ Das Verfahren ist inkrementell.
+ Durch die sehr starke Verteilung des Wissens hat das Verfahren eine sehr gute Verallgemeinerungsfähigkeit.
+ Es werden irrelevante Kriterien entdeckt.

Nachteile

- Es wird eventuell nicht das globale Minimum der Fehlerfunktion gefunden.

3.2.6 Bayessches Lernen

Ein statistischer Ansatz zur Akquisition einer Bewertungsfunktion beruht auf *Bayesschen Klassifikatoren*, die hauptsächlich in der Mustererkennung eingesetzt werden [7]. Dabei wird die Menge der Alternativen $\mathcal{A}^5$ in verschiedene (endlich viele) *Klassen* eingeteilt, wobei jede Klasse einer Bewertung entspricht. Jede Klasse K_i erhält eine *Zugehörigkeitsfunktion* g_i, die für eine Alternative a die bedingte Wahrscheinlichkeit $P(K_i|a)$ berechnet, daß a zur Klasse K_i gehört. Eine Alternative a wird dann mit derjenigen Bewertung (Klasse) versehen, deren Wert der Zugehörigkeitsfunktion am größten ist:

$$b(a) = i_{max} \quad \text{wobei} \quad g_{i_{max}}(a) = \max\{g_1(a), \ldots, g_n(a)\}$$

Die Zugehörigkeitsfunktion $g_i : \mathcal{A} \mapsto [0,1]$ für eine Klasse K_i ergibt sich aus dem *Satz von Bayes*:

$$g_i(a) = P(K_i|a) = \frac{p(a|K_i)P(K_i)}{p(a)} \quad \text{mit} \quad p(a) = \sum_j p(a|K_i)P(K_i)$$

wobei $P(K_i)$ die *a-priori*-Wahrscheinlichkeit für die Klasse i und p die *Wahrscheinlichkeitsdichte* für die Alternativen ist.

Die benötigten (exakten) Wahrscheinlichkeitsverteilungen stehen in der Regel nicht zur Verfügung und müssen anhand von Trainingsdaten, einer Menge von Alternativen mit ihren Bewertungen (Klassen), approximiert werden. Beispielsweise wird in [15] eine multivariate *Normalverteilung* der Daten angenommen, deren Parameter geschätzt werden.

Die Verwendung des Satz von Bayes erlaubt eine optimale Klassifikation in dem Sinn, daß die Fehlerwahrscheinlichkeit bei der Klassifizierung (Bewertung) bezüglich der approximierten Verteilungen minimal ist [7]. Die Güte der Bewertung entspricht also der Approximationsgüte der Wahrscheinlichkeitsverteilungen.

Verwendet man zwei Klassen K_+ ("gute" Alternativen) und K_- ("schlech-

wobei der Nenner $p(a)$ lediglich eine Skalierung darstellt und weggelassen werden kann.

Ein System, das eine Bewertungsfunktion auf diese Weise akquiriert ist BILL [15]. Dabei handelt es sich um ein Othello-Spielprogramm.

Vorteile

+ Es können "beliebige" Funktionen gelernt werden.
+ Korrelationen zwischen Bewertungskriterien werden berücksichtigt (in [15] wird dazu eine *Kovarianzmatrix* berechnet).
+ Es werden irrelevante Kriterien entdeckt.

Nachteile

- Die benötigten Wahrscheinlichkeitsverteilungen (bzw. deren Parameter) müssen anhand von Trainingsdaten geeignet approximiert werden.
- Das Verfahren ist nicht inkrementell.

3.2.7 Diskussion

Die Güte der Patche hängt wesentlich vom Zustand des Parsers (siehe Abschnitt 4.2.3) ab. Der funktionale Zusammenhang zwischen dem Parserzustand und den Prioritäten der Patche ist möglicherweise sehr komplex. Deshalb sollte das verwendete Lernverfahren nicht nur die Parameter einer vorgegebenen Funktion verändern, sondern die Möglichkeit bieten, "beliebige" Funktionen zu lernen.

Da keine optimalen Beispielparsläufe[6] als Trainingsdaten zur Verfügung stehen, sollte das Lernverfahren inkrementell sein, um die Bewertungsfunktion *sukzessiv* verbessern zu können. Der Einsatz eines nicht inkrementellen Verfahrens wäre ebenso möglich, was jedoch einen wesentlich höheren Trainingsaufwand zur Folge hätte.

Der Zustand des Parsers, von dem die Bewertung der Patche abhängt, wird durch die Anzahl von Patchen der verschiedenen Sorten beschrieben. Unter den Sorten bestehen in der Regel Korrelationen, die das Lernverfahren erkennen und entsprechend handhaben sollte, da es sonst zur *Übergewichtung* einiger Komponenten des Parserzustands kommen kann.

[6] Ein optimaler Parslauf führt auf direktem Weg zu den "guten" Parsen; Patche, die nicht zu diesen Parsen beitragen, werden auch nicht aus der Agenda ausgewählt. In manchen Domänen lassen sich optimale Suchverläufe eventuell "von Hand" führen. Dies ist jedoch beim Parsen von Graphen zu aufwendig.

Möglicherweise stellt sich bei einigen Sorten heraus, daß diese für die Bewertung irrelevant sind, was ebenfalls vom Lernverfahren berücksichtigt werden sollte.

Das Lernverfahren muß natürlich auch in der Lage sein, die Trainingsdaten auch auf nicht vorgelegte Beispielparse zu verallgemeinern.

Für die Realisierung "beliebiger" Funktionen kommen sowohl das Backpropagation-Verfahren als auch das Bayessche Lernen in Frage. Jedoch eignet sich das konnektionistische Backpropagation-Verfahren für die Erkennung von Korrelationen und irrelevanter Kriterien sowie für die Verallgemeinerung der Trainingsdaten wegen der ausgeprägten Verteilung des Wissens in besonderem Maße. Darüberhinaus ist das Verfahren inkrementell.

Als Konsequenz aus diesen Überlegungen soll für die Akquisition der Bewertungsfunktion zur Steuerung von GraPaKL das Backpropagation-Verfahren verwendet werden. Der Tatsache, daß Backpropagation eventuell in einem lokalen Minimum der Fehlerfunktion hängen bleibt, kann durch entsprechende Maßnahmen beim Training begegnet werden, so daß trotzdem ein dem globalen Minimum sehr nahe kommendes lokales Minimum gefunden wird.

Kapitel 4

Eine Heuristikkomponente für GraPaKL

In diesem Kapitel wird eine Komponente zur heuristischen Steuerung von GraPaKL vorgestellt. Ziel der heuristischen Steuerung ist es, daß "gute" Parse zuerst gefunden werden.

In Abschnitt 2.3 wurden zwei Ansatzpunkte zur heuristischen Steuerung von GraPaKL angesprochen. Während die Steuerung über die Ordnung der Rollenspezifikationen in den Produktionen bereits in GraPaKL integriert ist, wird in diesem Kapitel die heuristische Steuerung über die Ordnung der Patche in der Agenda betrachtet. Das hierzu benötigte Kontrollwissen wird durch ein Lernverfahren akquiriert werden.

Zunächst wird die Architektur der Heuristikkomponente vorgestellt. Anschließend werden das Bewertungs- und das Lernmodul im einzelnen beschrieben. Schließlich werden experimentelle Ergebnisse hinsichtlich der Eignung der hier vorgestellten Heuristikkomponente, soweit sie vorliegen, dargelegt.

4.1 Architektur

Die Heuristikkomponente besteht aus dem *Bewertungsmodul* und dem *Lernmodul*. Das Bewertungsmodul steuert die Suche des Parsers, indem er mittels einer Bewertungsfunktion Prioritäten an die Patche in der Agenda vergibt. Dies entspricht einer *Best-First*-Suche; der `Choose`-Operator wählt gemäß der Prioritäten das jeweils vielversprechendste Patch aus der Agenda aus. Das Lernmodul ist für die Akquisition der Bewertungsfunktion zuständig.[1]

[1] Im *Lernmodus* sind beide Module aktiv, während im *Arbeitsmodus* lediglich das Bewertungsmodul arbeitet.

Das Bewertungsmodul ist in einen *statischen* und einen *dynamischen* Teil aufgespalten (Abb. 4.2). Dies soll im folgenden motiviert werden.

Eine rein *statische* Bewertung (Abb. 4.1 a)), welche die Patche *einmalig* mit einer Priorität versieht, bevor diese in die Agenda eingetragen werden, ist zu schwach, da die Güte eines Patches hinsichtlich einiger Kriterien *situationsabhängig*, d.h. abhängig vom aktuellen Zustand des Parsers ist.[2] Die Bewertung der Patche würde trotz fortlaufender Änderung des Parserzustands nicht mehr revidiert werden und die Suche somit u.U. in eine falsche Richtung lenken. Dieses Problem wird gelöst, indem die Güte der Patche

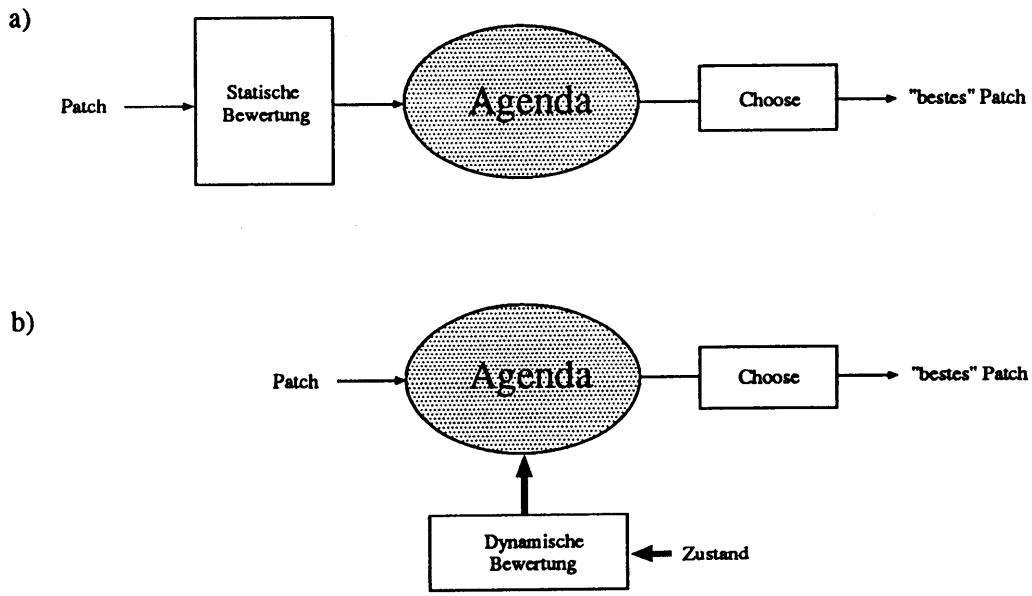


Abbildung 4.1: a) Statische vs. b) dynamische Bewertung

hinsichtlich der vom Parserzustand abhängigen Kriterien *dynamisch* bewertet wird (Abb. 4.1 b)), d.h. die Bewertung wird nach jeder Zustandsänderung gemäß des neuen Zustands *aktualisiert*.[3]

Deshalb wird eine *hybride* Vorgehensweise realisiert, indem sowohl eine statische als auch eine dynamische Bewertung der Patche vorgenommen wird

[2] In Kapitel 3 wurde bereits erwähnt, daß unter dem *Parserzustand* die bereits gefundenen kompletten Patche (erkannte Features) verstanden werden soll; dies wird in Abschnitt 4.2.3 formal definiert.

[3] Dies läßt sich variieren, indem man die Patche in der Agenda nur nach jeder k-ten Zustandsänderung neu bewertet.

(Abb. 4.2), wobei für beide Teilbewertungen geeignete Kriterien zu wählen sind.

Die *statische Priorität* wird mit der Erzeugung des Patches, also bevor es in die Agenda eingetragen wird, durch die Funktion ϕ_{stat} berechnet und dem Patch zugeordnet, d.h. in die Datenstruktur des Patches eingetragen. Die *dynamische Priorität* der Patche wird durch die Funktion ϕ_{dyn} in Abhängigkeit des aktuellen Parserzustands ständig neu berechnet. Durch eine geschickte Strukturierung der Agenda wird erreicht, daß dabei nicht jedes Patch einzeln betrachtet werden muß (siehe Abschnitt 4.2.4). Dies ist notwendig, weil die Anzahl der Patche in der Agenda in der Regel sehr groß ist.

Die Akquisition der Bewertungsfunktion ist Aufgabe des Lernmoduls. Dieses erhält von einem Experten Beispielparse und extrahiert aus den während der zugehörigen Parsläufe getroffenen Auswahlentscheidungen Trainingsinstanzen für den Lernalgorithmus, der die Bewertungsfunktion verändert (Abb. 4.2). Die Parsläufe, aus denen gelernt wird, lassen sich als Fälle (Episoden) auffassen. Jedoch werden diese nach dem Lernprozeß *nicht* mehr aufbewahrt, so daß die heuristische Steuerung nicht als fallbasiert bezeichnet werden kann (vgl. Abschnitt 3.1.4).

In Abschnitt 2.3 wurde bereits gesagt, daß die Steuerung des Parsers über die Ordnung der Rollenspezifikationen in den Produktionen vollständig der Kontrolle des Benutzers unterliegt; das Kontrollwissen ist explizit. Bei der Steuerung des Parsers über die Ordnung der Patche in der Agenda mit der hier vorgestellten Architektur ist das Kontrollwissen in den Parametern der Bewertungsfunktion kodiert, es ist also *implizit*. Die Kontrolle des Benutzers beschränkt sich somit auf das Vorlegen geeigneter Beispielparse, aus denen die Bewertungsfunktion akquiriert werden soll. Der Benutzer kann den Parser also nur *mittelbar* (über das Lernmodul) steuern; die konkrete Bewertungsfunktion bleibt dem Benutzer verborgen.

An dieser Stelle sei hervorgehoben, daß diese Architektur *unabhängig* von der zugrundeliegenden Graphgrammatik ist und somit auch in anderen Domänen (z.B. Analyse von Molekülstrukturen) einsetzbar ist. Darüberhinaus ist es denkbar, diese Architektur auch auf agenda-basierte Chart-Parser für Stringgrammatiken (z.B. Parsing natürlicher Sprache) zu übertragen.

4.2 Das Bewertungsmodul

Dieser Abschnitt beschreibt das Modul, das die Bewertung der Patche in der Agenda vornimmt. Es besteht aus der statischen und der dynamischen Teilbewertung. Letztere hat einen wesentlich größeren Einfluß auf das Suchverhalten des Parsers, da sie sich situationsabhängig verhält. Deshalb wird

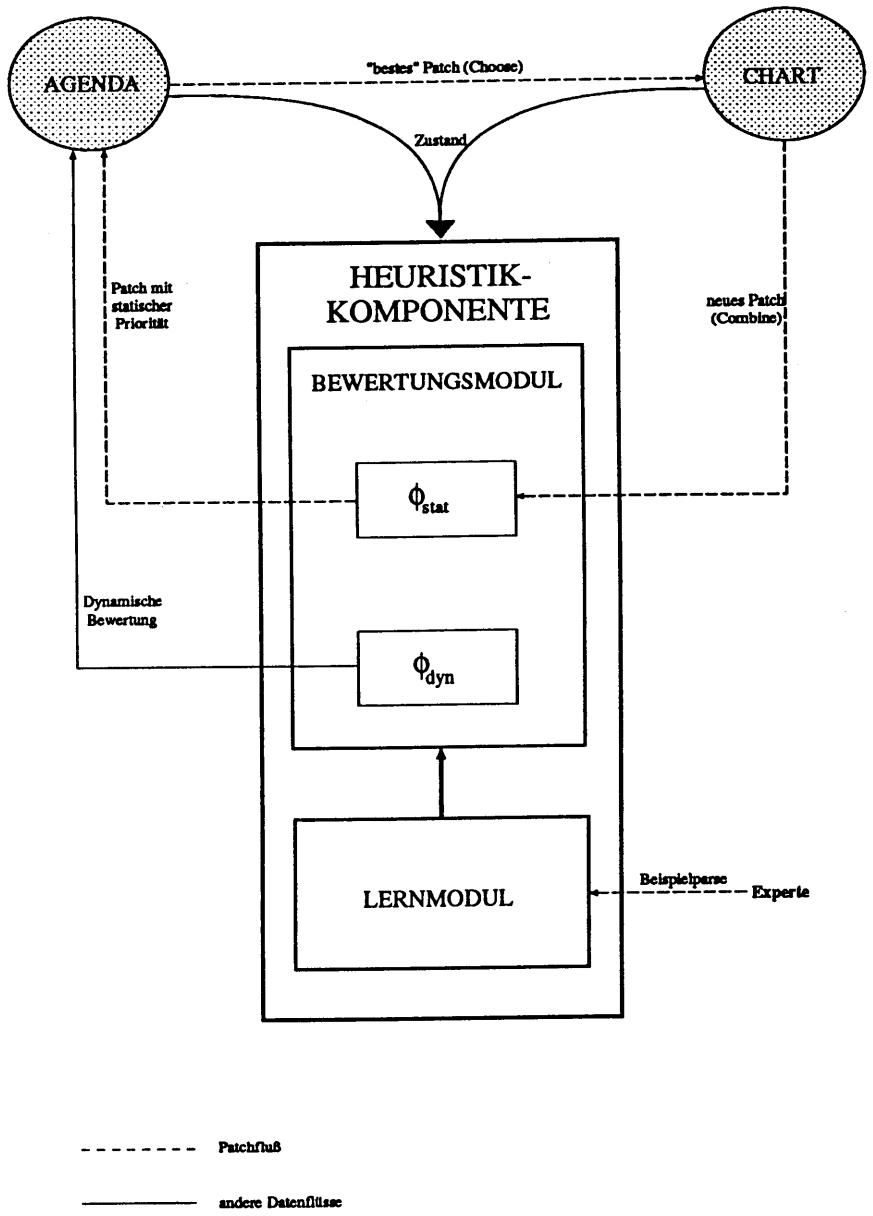


Abbildung 4.2: Die Architektur der Heuristikkomponente für GraPaKL

Teilbewertung die *Sekundärsteuerung* darstellt.[4]

4.2.2 Bewertungskriterien

Die kompletten Patche erhalten pauschal die höchste Priorität. Um eine Bewertung der partiellen Patche vornehmen zu können, müssen zunächst neben dem Zustand des Parsers zusätzliche *Bewertungskriterien* für die statische und dynamische Teilbewertungen ausgewählt werden. Patche lassen sich prinzipiell durch verschiedene Eigenschaften beschreiben, aus denen Bewertungskriterien ausgewählt werden müssen. Dies sind insbesondere

- die Sorte des Patches,
- die Sorte der zu besetzenden Rolle,
- die Identifikation der zugehörigen Produktion,
- die Identifikation des Patches,
- der Anteil des Patches am Eingabegraphen ("Span")[6]
- der Anteil der bereits besetzten Rollen an der Gesamtrollenanzahl der zugehörigen Produktion.

Durch die heuristische Steuerung wird die Anzahl der insgesamt bis zum Finden des ersten Parse erzeugten Patche reduziert. Um jedoch insgesamt einen Laufzeitgewinn zu erzielen, ist darauf zu achten, daß der zusätzliche Berechnungsaufwand für die Bewertung und Sortierung der Patche in der Agenda diese Ersparnis nicht wieder egalisiert oder gar übersteigt. Diese Überlegung hat Konsequenzen für die Auswahl der Bewertungskriterien. Diese sollten demnach so ausgewählt werden, daß sich damit einerseits die Anzahl der insgesamt erzeugten Patche (bis zum Finden des ersten "guten" Parse) erheblich reduzieren läßt und andererseits der Berechnungsaufwand so gering ist, daß ein genügend hoher Laufzeitgewinn erzielt wird.

Deshalb soll die dynamische Bewertung entweder bezüglich der *Sorte* des Patches oder bezüglich der *Identifikation der Produktion*, deren Instanz das Patch ist, erfolgen. Dies bewirkt, daß gemäß der aktuellen Situation (Zustand) Patche der momentan benötigten Sorte (bzw. Produktion) ausgewählt bzw. angewendet werden und daß Hypothesen, deren Sorte (bzw. Produktion) sich während des Parsing-Prozesses als unbrauchbar erweist, aufgegeben werden. Eine exaktere Steuerung läßt sich mit der Verwendung der Identifikation der Produktion als Bewertungskriterium erreichen, da jede Sorte im allgemeinen durch *mehrere* Produktionen beschrieben ist. Dem steht jedoch der höhere Berechnungsaufwand gegenüber, so daß sich eventuell keine Verbesserung ergibt. Welches Kriterium letztendlich das bessere ist, muß *experimentell* untersucht werden. Im folgenden wird als Kriterium für die

[6] Der *Span* eines Patches mißt den (prozentualen) Anteil der durch das Patch bereits erfaßten terminalen Knoten an der Gesamtknotenanzahl des (terminalen) Eingabegraphen.

dynamische Bewertung die Sorte des Patches verwendet, kann aber *ohne* wesentliche Änderungen gegen die Identifikation der Produktion ausgetauscht werden.

Die statische Bewertung soll bezüglich des (prozentualen) *Anteils der schon belegten Rollen* und des *"Span"* des Patches vorgenommen werden. Damit wird erreicht, daß unter Patchen gleicher dynamischer Priorität diejenigen ausgewählt werden, die am weitesten "fortgeschritten" sind, damit der erste Parse im Rahmen der dynamischen Bewertung so früh wie möglich gefunden werden kann.

Die Werte der Bewertungskriterien müssen in der Datenstruktur der Patche für das Bewertungsmodul zur Verfügung gestellt werden. Während die Sorte des Patches bereits in der Datenstruktur vorhanden ist, muß die Datenstruktur für die beiden anderen entsprechend erweitert werden. Man beachte, daß der Aufwand für die Berechnung dieser Werte *gering* (konstant) ist. Für die Anzahl der besetzten Rollen muß lediglich der analoge Wert des partiellen "Elternpatches" inkrementiert (und normiert) werden, und der "Span" berechnet sich aus der Summe der analogen Werte der "Elternpatche".[7] Im folgenden kann davon ausgegangen werden, daß diese Werte mit Hilfe der Selektorfunktionen *sort*, *span* und *comp*[8] aus der Patch-Datenstruktur extrahiert werden können.

4.2.3 Die Bewertungsfunktion

Zunächst ist eine geeignete Menge von Prioritätswerten zu wählen, in die die Patche durch die Bewertungsfunktion abgebildet werden. Ein Prioritätswert soll ein Vektor $(dyn, stat) \in [0,1] \times [0,1]$ sein, wobei *dyn* der dynamischen und *stat* der statischen Priorität entspricht. Des weiteren muß eine Ordnung auf den Prioritätswerten definiert werden, damit eine *Semantik* für die Relation "besser als" festgelegt werden kann, so daß der `Choose`-Operator das jeweils vielversprechendste Patch aus der Agenda auswählen kann.

Definition 4.1 *Die* **Prioritätsordnung** $<_{\mathcal{V}}$: $\mathcal{V} \times \mathcal{V} \mapsto \{0,1\}$ *auf der* **Menge der Prioritätsvektoren** $\mathcal{V} \subseteq [0,1] \times [0,1]$ *ist definiert durch*

$$(dyn_1, stat_1) <_{\mathcal{V}} (dyn_2, stat_2)$$
$$gdw.$$
$$dyn_1 < dyn_2 \text{ oder } (dyn_1 = dyn_2 \text{ und } stat_1 < stat_2)$$

[7] Treten in der Grammatik Überlappungsbeziehungen auf, so ist bei der "Span"-Berechnung darauf zu achten, daß einzelne Knoten des Eingabegraphen nicht mehrfach gezählt werden.

[8] Dabei steht *comp* für "degree of completeness", den Anteil der bereits belegten Rollen.

wobei $(dyn_1, stat_1), (dyn_2, stat_2) \in \mathcal{V}$ *und* $<$ *die übliche kleiner-Relation auf* IR *ist. Zwei Prioritätsvektoren* $v, v' \in \mathcal{V}$ *sind gleich, d.h.* $v =_\mathcal{V} v'$*, falls sie in beiden Komponenten übereinstimmen.*

Man beachte, daß diese Ordnung *total* ist. Die Definition der Ordnung $<_\mathcal{V}$ spiegelt den Sachverhalt wider, daß die Bewertung der Patche zweistufig ist, wobei die dynamische Teilbewertung der Primärsteuerung entspricht.

Im folgenden wird die Bewertungsfunktion definiert. Sie setzt sich aus der statischen sowie der dynamischen Bewertungsfunktion zusammen, die vorab definiert werden sollen.

Die dynamische Bewertungsfunktion, die bezüglich der Sorte des Patches in Abhängigkeit des (aktuellen) Parserzustands erfolgt, basiert auf der *Sortenprioritätsfunktion* χ, die dem aktuellen Zustand einen Vektor zuordnet, der die (aktuelle) Priorität jeder Sorte enthält. χ bildet den Kern der gesamten Bewertungsfunktion und soll durch das Lernmodul akquiriert werden.

[redacted]

Die statische Bewertungsfunktion ist eine Linearkombination aus dem Anteil der belegten Rollen und dem "Span" des Patches, wobei die Koeffizienten als Parameter vorgesehen sind, die geeignet zu wählen sind. Sie bestimmt, das (unter Patchen gleicher dynamischer Priorität) solche Patche vorgezogen werden sollen, die am weitesten "fortgeschritten" sind. Gegenüber der dynamischen Bewertungsfunktion soll hier kein Lernverfahren eingesetzt werden; die Parameter lassen sich auch "von Hand" hinreichend gut wählen.[10]

Definition 4.4 *Sei $\mathcal{PP}$ die Menge der partiellen Patche. Die* **statische Patchbewertungsfunktion** *ϕ_{stat}: $\mathcal{PP} \mapsto]0,1[$ ist definiert durch*

$$\phi_{stat}(p) = \frac{1}{a+b} \cdot (a \cdot comp(p) + b \cdot span(p))$$

wobei *a und b natürliche Zahlen und*
comp, span : $\mathcal{PP} \mapsto]0,1[$ die **Selektorfunktionen** *für den Anteil der belegten Rollen bzw. den "Span" des Patches sind.*

Nun kann die gesamte Bewertungsfunktion definiert werden. Sie gibt kompletten Patchen höchste Priorität und partielle Patche werden den beiden Teilbewertungen unterzogen.

Definition 4.5 *Sei $\mathcal{P}$ die Menge der Patche und $\mathcal{Z}$ die Menge der Parserzustände. Die* **Patchbewertungsfunktion** *ϕ: $\mathcal{P} \times \mathcal{Z} \mapsto \mathcal{V}$ ist definiert durch*

$$\phi(p,z) = \begin{cases} \top & falls\ p\ komplett \\ (\phi_{dyn}(p,z), \phi_{stat}(p)) & falls\ p\ partiell \end{cases}$$

wobei $\top = (1,1)$ das **Topelement** *der Ordnung der Prioritätsvektoren ist.*

Die kompletten Patche bekommen alle die gleiche (höchste) Priorität. Die Bewertung ist somit *nicht* eindeutig, d.h. es gibt Patche, die die gleiche Priorität erhalten.

Schließlich kann die *Semantik* für "besser als" festgelegt werden, so daß der `Choose`-Operator das jeweils beste Patch aus der Agenda auswählen kann.

Definition 4.6 *Sei $\mathcal{P}_A$ die Menge der Patche, die sich (momentan) in der*

[9] Die Projektion π_{s_i} extrahiert lediglich die Priorität der Sorte $s_i \in \mathcal{S}$ aus dem Sortenprioritätsvektor: $\pi_{s_i}(prio_{s_1}, \ldots, prio_{s_i}, \ldots, prio_{s_n}) = prio_{s_i}$.

[10] Es ist aber durchaus denkbar, für die Justierung der Koeffizienten ebenfalls eines der in Kapitel 3 vorgestellten Lernverfahren für lineare Bewertungsfunktionen einzusetzen. Dies könnte eventuell in anderen Domänen interessant sein, wo mehrere *verschiedenartige* Kriterien zu berücksichtigen sind.

Agenda befinden. Ein Patch $p_1 \in \mathcal{P}_A$ *heißt (bezüglich des Parserzustands z)* **besser** *als ein Patch* $p_2 \in \mathcal{P}_A$*, falls* $\phi(p_2, z) <_V \phi(p_1, z)$.
Ein Patch $p \in \mathcal{P}_A$ *heißt (bezüglich des Parserzustands z)* **bestes Patch**, *falls für alle anderen Patche* $p' \in \mathcal{P}_A$ *gilt:* $\phi(p, z) \not<_V \phi(p', z)$.

Man beachte, daß es wegen der Uneindeutigkeit der Ordnung mehrere beste Patche bezüglich eines Parserzustandes geben kann.

4.2.4 Effiziente Implementierung der Patchbewertung

Die dynamische Bewertung bewirkt eine ständige Änderung der Ordnung auf der Agenda. Um den Berechnungsaufwand für die Bewertung und Sortierung der Patche und die Choose-Operation möglichst gering zu halten, wird eine *Strukturierung* der Agenda vorgenommen.

Strukturierung der Agenda

Zunächst werden komplette und partielle Patche separat gehalten, da komplette Patche pauschal höhere Priorität als partielle Patche haben. Die kompletten Patche werden in einer (ungeordneten) Liste gespeichert, da sie alle die gleiche (höchste) Priorität erhalten und somit die Auswahlreihenfolge keine Rolle spielt.

Um bei der dynamischen Bewertung, die nach jeder (k-ten) Zustandsänderung aktualisiert wird, nicht jedes Patch einzeln bewerten zu müssen, werden die Patche nach ihrer Sorte separiert. Somit müssen nur die verschiedenen Sorten betrachtet werden, nicht die Patche selbst. Bei der Auswahl des besten Patches kann dann einfach über die (momentan) beste Sorte auf die Menge der Patche dieser Sorte zugegriffen werden.

Die Separierung läßt sich beispielsweise als Hashtabelle realisieren, wobei die Selektorfunktion *sort* als Hashfunktion dient.[11] Unter einem bestimmten Hashschlüssel, also einer gewissen Sorte, steht dann eine Liste aller in der Agenda befindlichen partiellen Patche dieser Sorte. Diese Listen sind gemäß der *statischen* Priorität (absteigend) sortiert, so daß der Choose-Operator schnell auf das Patch mit der insgesamt höchsten Priorität zugreifen kann.

[11] Werden mehrere Kriterien zur dynamischen Bewertung verwendet, so muß nach mehreren Kriterien separiert werden; dazu wird eine mehrdimensionale Datenstruktur benötigt. In jeder Dimension werden die Patche nach den Ausprägungen eines der Kriterien getrennt. Es empfiehlt sich jedoch aus Komplexitätsgründen die Anzahl der Kriterien möglichst klein zu halten.

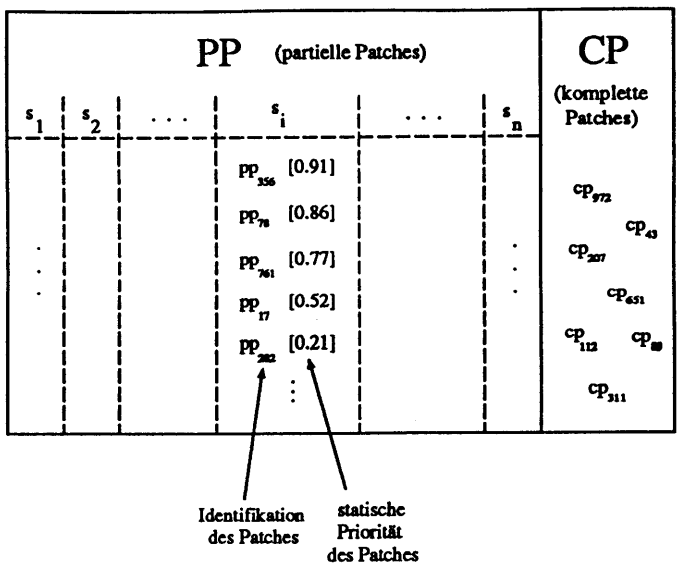


Abbildung 4.3: Die Struktur der Agenda

Die Struktur der Agenda veranschaulicht Abbildung 4.3. Folgender Algorithmus beschreibt das Einfügen eines Patches in die strukturierte Agenda.

Algorithmus 4.1 (Einfügen in die Agenda)
Sei im folgenden $\mathcal{CP}_{\mathcal{A}}$ die Menge der kompletten Patche und $\mathcal{PP}_{\mathcal{A}}(s)$ die Menge der partiellen Patche der Sorte s, die sich (momentan) in der Agenda befinden.

Eingabe: *Neu erzeugtes Patch 'newpatch'*

begin
  if *newpatch is complete*
    then $\mathcal{CP}_{\mathcal{A}} \leftarrow \mathcal{CP}_{\mathcal{A}} \cup \{newpatch\}$;
    else
      compute static priority $\phi_{stat}(newpatch)$;
      insert newpatch in $\mathcal{PP}_{\mathcal{A}}(sort(newpatch))$ *wrt. its static priority;*
end

Wegen der Uneindeutigkeit der Bewertung können verschiedene Patche die gleiche statische Priorität haben. In diesem Fall ist die Reihenfolge der Patche

gleicher statischer Priorität beliebig.

Durch diese Strukturierung der Agenda müssen nach einer Zustandsänderung nicht *alle* Patche neu bewertet und sortiert werden; es werden lediglich die Sorten mittels χ bewertet und entsprechend sortiert. Dies reduziert den Berechnungsaufwand *erheblich*, da die (variable) Anzahl der Patche in der Agenda im allgemeinen um ein Vielfaches höher ist als die (konstante) Anzahl der Sorten.

Implementierung des Choose-Operators

Nun kann der Choose-Operator, der das beste Patch aus der Agenda auswählen soll, auf die Struktur der Agenda abgestimmt werden.

Algorithmus 4.2 (Choose)
Seien im folgenden $\mathcal{CP}_\mathcal{A}$ und $\mathcal{PP}_\mathcal{A}(s)$ wie in Algorithmus 4.1.

Ausgabe: *ein bestes Patch aus der Agenda*

begin
  if $\mathcal{CP}_\mathcal{A} \neq \emptyset$
    then return *any* cp $\in \mathcal{CP}_\mathcal{A}$;
    else
      compute (dynamic) priority of all sorts $s_1, ..., s_n$ *by* χ;
      compute decreasing order $s_{i_1}, ..., s_{i_n}$ *wrt. their priority;*
      currentsort $\leftarrow s_{i_1}$;
      while $\mathcal{PP}_\mathcal{A}(currentsort) = \emptyset$ **do**
        sort $\leftarrow succ(currentsort)$;
      return *first (= one of the best)* $pp \in \mathcal{PP}_\mathcal{A}(currentsort)$;
end

Sind komplette Patche in der Agenda vorhanden, so wählt er irgendeins davon aus. Andernfalls stößt er die dynamische Bewertung, also die Sortenprioritätsfunktion χ an, welche aus dem aktuellen Zustand die Sortenprioritäten berechnet. Nun wählt er das erste, also das statisch beste Patch aus der Patchliste der Sorte mit der höchsten Priorität, falls diese nicht leer ist, aus. Ansonsten betrachtet er die Patchliste der Sorte mit der zweithöchsten Priorität. Ist ebenfalls kein Patch dieser Sorte vorhanden, so betrachtet er die

Wegen der Uneindeutigkeit der dynamischen Bewertung kann es vorkommen, daß verschiedene Sorten die gleiche Priorität bekommen. In diesem Fall ist die Reihenfolge der Sorten gleicher Priorität beliebig (vgl. Algorithmus 4.1).

Möglicherweise ist trotz hoher Priorität einer Sorte kein Patch dieser Sorte in der Agenda vorhanden. Die Aussage der Heuristik ist auch eher die, daß *falls* Patche dieser Sorte in der Agenda vorhanden sind, es günstig ist, diese auszuwählen. Jedoch ist die Wahrscheinlichkeit, daß der `Choose`-Operator mehrere Sorten "durchprobieren" muß bis er ein Patch findet eher gering, da die Berechnung der Sortenprioritäten aus vielen Parsläufen *gelernt* wird.

4.3 Das Lernmodul

In diesem Abschnitt wird das Lernmodul vorgestellt. Dieses soll die dynamische Bewertungsfunktion, also die Sortenprioritätsfunktion χ akquirieren. Der Lerngegenstand ist also eine *Abbildung* (χ), die einer gewissen Situation (Parserzustand) Sortenprioritäten zuordnet, so daß die richtige Auswahlentscheidung in dieser Situation getroffen wird. Da keine optimalen Parsläufe

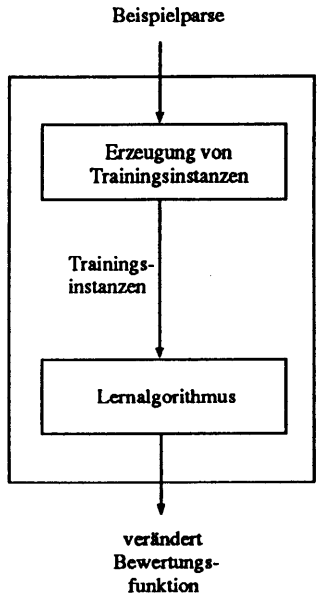


Abbildung 4.4: Interne Struktur des Lernmoduls

zur Akquisition von χ zur Verfügung stehen, wird *inkrementell* vorgegangen.

Dazu werden anfangs Werkstücke *ohne Heuristik* geparst und dem Lernmodul "gute" Parse vorgelegt, um aus den Auswahlentscheidungen, die zu diesen Parsen geführt haben, Trainingsinstanzen für den Lernalgorithmus zu erzeugen, der die initiale Sortenprioritätsfunktionfunktion verändert. Mit dieser ersten Heuristik werden dann die Werkstücke erneut geparst und dem Lernmodul "gute" Parse (die nun auf kürzerem Wege gefunden wurden) präsentiert, welches χ weiter verändert. Es werden so viele Iterationen durchgeführt, bis eine genügend große Verbesserung der Suchleistung erreicht ist.

Hier tritt das Problem auf, welche Entscheidungen (Auswahl der Patche) während der Suche zu einem "guten" Parse beigetragen haben und welche nicht. Dies ist das sogenannte *Credit Assignment Problem* [23] und wird im nächsten Abschnitt behandelt. Die Struktur des Lernmoduls veranschaulicht Abbildung 4.4.

In Kapitel 3 wurden verschiedene Lernverfahren diskutiert und festgestellt, daß sich Backpropagation (Abschnitt 3.2.5) am besten zur Akquisition der dynamischen Bewertungsfunktion eignet. Somit wird χ als Backpropagationnetz realisiert (Abb. 4.5), dessen Gewichte durch den Backpropagationalgorithmus eingestellt werden.[12]

4.3.1 Credit Assignment und Erzeugung von Trainingsinstanzen

Aus einer Menge vorgelegter Beispielparse ist eine Menge von Trainingsinstanzen für den Lernalgorithmus zu erzeugen. Die Trainingsinstanzen sind Ein-/Ausgabepaare (in, out), wobei *out* die gewünschte Ausgabe zur Eingabe *in* ist.

Da keine optimalen Parsläufe zur Verfügung stehen, ist man mit dem *Credit Assignment Problem* konfrontiert. Es muß festgestellt werden, welche während der Suche ausgewählten Patche zu einem vorgelegten Parse "beigetragen" haben, d.h. welche Patche zur Konstruktion des Patches, welches den Parse repräsentiert, benötigt wurden und welche nicht.

Um dies rückverfolgen zu können wird während des Parsvorgangs für *jedes* erzeugte Patch p (also auch für die Parse) eine Liste dessen "Vorfahren" $history(p)$ berechnet und in der Datenstruktur von p mitgeführt. Für ein erzeugtes Patch p läßt sich $history(p)$ berechnen durch

$$history(p) = \emptyset$$

[12] Man beachte, daß die Aktivität der Ausgabeneuronen (= dynamische Prioritäten der Patche) im Intervall]0,1[liegt, was der Definition der dynamischen Bewertungsfunktion ϕ_{dyn} entspricht (vgl. Definition 4.3).

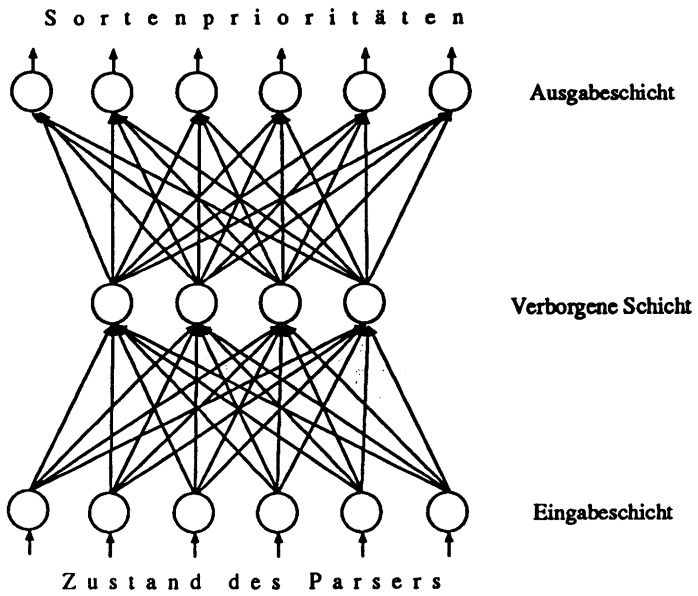


Abbildung 4.5: Realisierung der Sortenprioritätsfunktion χ als Backpropagationnetz

falls p eines der Patche ist, mit denen die Agenda initialisiert wird,

$$history(p) = history(cp) \cup \{cp\}$$

falls p eine Neuinstantiierung einer Produktion ist (d.h. cp belegt die erste Rolle von p und die restlichen Rollen sind noch frei) und

$$history(p) = history(cp) \cup history(pp) \cup \{cp\} \cup \{pp\}$$

falls p aus der Kombination zweier Elternpatche cp und pp entstanden ist.[13]

Hat ein Patch p zu einem vorgelegten Parse P beigetragen, d.h. $p \in history(P)$, so soll die Sorte von p bezüglich des Parserzustands z, in dem p ausgewählt wurde eine *hohe* und alle anderen Sorten eine *niedrige* Bewertung bekommen.[14] Es wird eine Trainingsinstanz (in, out) erzeugt mit

$$\begin{aligned} in &= z \\ out &= (0, \ldots, 0, 1, 0, \ldots, 0) \end{aligned}$$

[13]Dabei sind eventuelle Mehrfachvorkommen von Patchen, die durch die Überlappungsbeziehung entstehen, zu löschen.

[14]Der Parserzustand kann als *Erklärung* für die Auswahl eines Patches einer gewissen Sorte angesehen werden.

wobei *out* eine Eins (hohe Bewertung) für die Sorte des Patches und Nullen (niedrige Bewertung) für alle anderen Sorten enthält.[15] Dies läßt sich so interpretieren, daß z ein *positives* Beispiel für die Auswahl der Sorte von p ist und ein *negatives* für die Auswahl aller anderen Sorten.

Man erhält somit aus den vorgelegten Beispielparsen eine Menge $\mathcal{T}$ von Trainingsinstanzen, mit der das Backpropagationnetz trainiert wird.

4.3.2 Anmerkungen zum Training

Das Training besteht aus mehreren *Epochen*. In jeder Epoche werden alle Trainingsinstanzen aus $\mathcal{T}$ dem Backpropagationalgorithmus genau einmal vorgelegt. Es werden so viele Epochen durchgeführt, bis der Fehler auf das gewünschte Maß reduziert ist.

Netzarchitektur

Je größer die Anzahl der verborgenen Schichten des Netzes bzw. die Anzahl der Neuronen in den verborgenen Schichten ist, desto mehr Speicherkapazität hat das Netz und desto mehr läßt sich der Lernprozeß als "Auswendiglernen" charakterisieren. Deshalb ist eine zu große Zahl verborgener Schichten bzw. Neuronen nicht wünschenswert, da sonst keine ausreichende *Verallgemeinerung* der Trainingsinstanzen stattfindet und somit der Erfolg der gelernten Heuristik auf die Werkstücke beschränkt ist, die zum Training verwendet wurden. Ziel ist es aber, daß der Suchaufwand auch für noch nicht gesehene Werkstücke reduziert wird. Jedoch darf die Anzahl der verborgenen Neuronen auch nicht zu klein sein, da sonst kein adäquater Lernerfolg eintritt.

Kodierung des Parserzustands

Um einen besseren Lerneffekt zu erreichen, soll der Parserzustand $z \in [0,1]^S$ nicht direkt als Netzeingabe verwendet werden, sondern so kodiert werden, daß jede Komponente des Zustandsvektors auf mehrere Eingabeneuronen *verteilt* wird. Durch diese verteilte Repräsentation der Komponenten wird erreicht, daß wegen der höheren Anzahl der trainierbaren Verbindungen bzw. Gewichte die Abbildung schneller und besser gelernt werden kann.

Dabei ist jedoch darauf zu achten, daß die Kodierungsabbildung *topologieerhaltend* ist, d.h. daß "ähnliche" Zustände auch auf "ähnliche" Kodierungen abgebildet werden. Beispielsweise ist eine Binärkodierung ungeeignet, da z.B.

[15] Dabei wird der Parserzustand z *kodiert*, um ein besseres Lernverhalten zu erzielen (siehe Abschnitt 4.3.2).

die benachbarten Werte 7 und 8 in die völlig verschiedenen Muster 0111 und 1000 kodiert würden.

Deshalb soll die topologieerhaltende *Thermometerkodierung* [18] verwendet werden. Danach werden die Komponenten des Parserzustands gemäß der Kodierungsabbildung $\tau : [0,1] \mapsto [0,1]^n$ auf n Eingabeneuronen verteilt. Für $n = 5$ hat τ folgende Gestalt:

$$\tau(x) = \begin{cases} (\frac{x}{0.2}, 0, 0, 0, 0) & falls\ x \in [0, 0.2[\\ (1, \frac{x-0.2}{0.2}, 0, 0, 0) & falls\ x \in [0.2, 0.4[\\ (1, 1, \frac{x-0.4}{0.2}, 0, 0) & falls\ x \in [0.4, 0.6[\\ (1, 1, 1, \frac{x-0.6}{0.2}, 0) & falls\ x \in [0.6, 0.8[\\ (1, 1, 1, 1, \frac{x-0.8}{0.2}) & sonst \end{cases}$$

Mit der Wahl von n läßt sich das Ausmaß der Verteilung steuern.

Parameter des Backpropagationalgorithmus

Der Backpropagationalgorithmus hat mehrere Parameter, die geeignet zu wählen sind:

- Lernrate η
- Koeffizient α des Momentum-Terms
- Initialisierung der Gewichte
- Temperaturparameter T der Aktivierungsfunktion

Mit der Lernrate läßt sich die Größe der Gewichtsänderung, d.h. wie weit in Richtung des steilsten Abstiegs im Fehlergebirge gegangen wird, steuern. Ist sie zu klein, so dauert das Training sehr lange; ist sie hingegen zu groß, so werden die Minima eventuell übersprungen.

Der Koeffizient des Momentum-Terms bestimmt den Einfluß der letzten Gewichtsänderung. Ist er zu klein, so "oszilliert" die Fehlerfunktion um ein (eventuell lokales) Minimum herum, erreicht dieses aber nicht; ist er zu groß, so wird eventuell in die falsche Richtung im Fehlergebirge gegangen.

Zu Beginn des Trainings müssen die Gewichte des Netzes *initialisiert* werden. Von diesen initialen Gewichten hängt der Lernerfolg ebenfalls ab, da bei einer unglücklichen Wahl dieser Anfangswerte eventuell nur ein unbrauchbares (lokales) Minimum der Fehlerfunktion E erreicht wird.

Der Temperaturparameter steuert den Verlauf der Aktivierungsfunktion und hat somit ebenfalls Einfluß auf das Training.

4.4 Experimentelle Ergebnisse

Im Rahmen dieser Arbeit konnten lediglich *einfache* Testläufe durchgeführt werden; die Auslotung der vollen Leistungsfähigkeit der Heuristikkomponente (vor allem des Backpropagation-Verfahrens) erfordert *umfangreiche* Testreihen, um eine optimale Wahl der Parameter (η, α, T und initiale Gewichte) und der Netzarchitektur (Anzahl und Stärke verborgener Schichten) zu treffen. Somit können die vorliegenden Ergebnisse lediglich die *prinzipielle* Eignung der Konzeption der Heuristikkomponente und der Wahl des Lern-

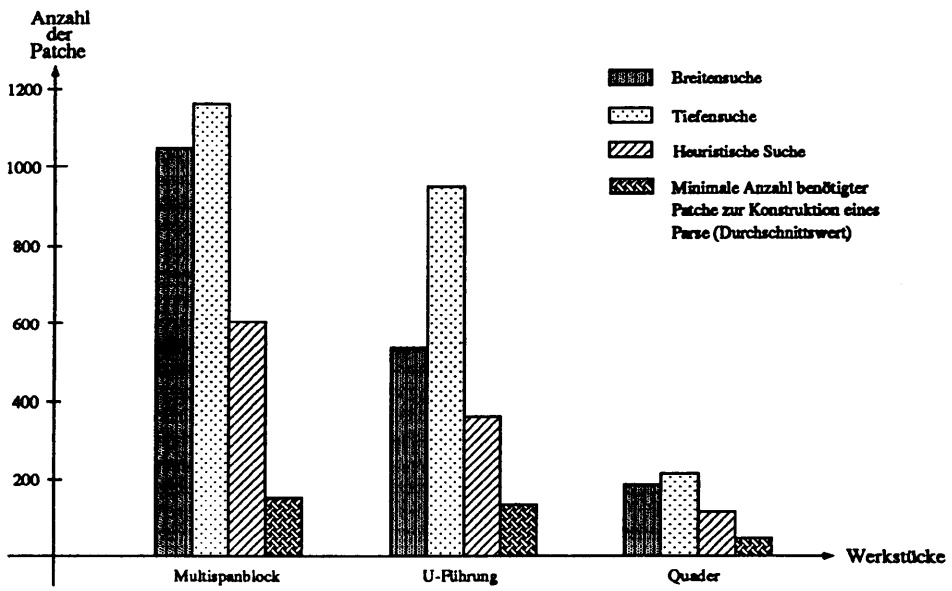


Abbildung 4.6: Vergleich des Suchaufwands für Breiten-, Tiefen- und heuristische Suche bei Frästeilen

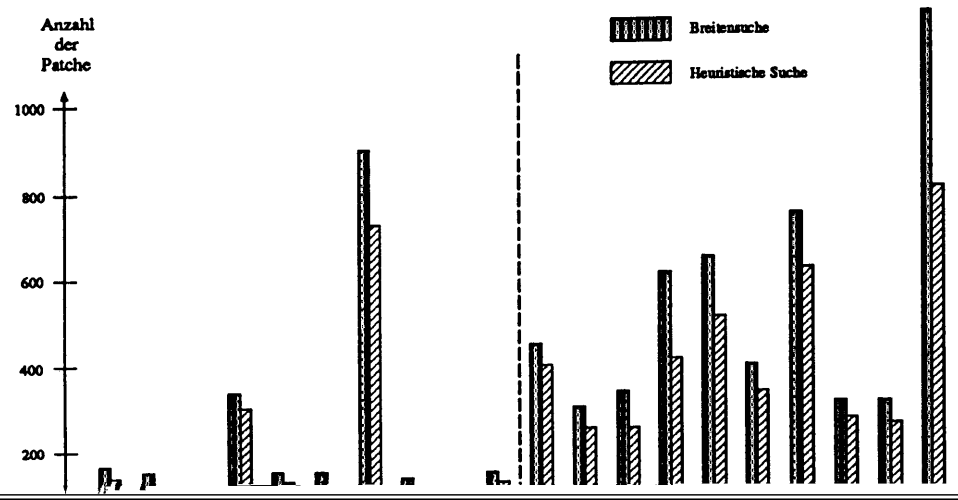
Experiment 2

Mit dem zweiten Experiment sollte getestet werden, ob sich die Reduktion des Suchaufwands auch auf Werkstücke, von denen *kein* Parse zum Lernen herangezogen wurde, überträgt.

Für dieses Experiment wurde die Drehteilgrammatik 'Turning' verwendet, weil bei dieser 20 Testwerkstücke zur Verfügung standen. Dabei wurden in jeder Iteration die ersten gefundenen Parse (bzw. die daraus erzeugten Trainingsinstanzen) von *nur* 10 dieser Werkstücke als Trainingsdaten genommen. Es wurde folgende Netzarchitektur für χ gewählt: 275 Eingabeneuronen (55 Sorten auf je 5 Neuronen verteilt), 10 verborgene Neuronen und 55 Ausgabeneuronen (für jede Sorte 1 Neuron).

Nach bereits 1 durchgeführten Iteration mit $\eta = 2$, $\alpha = 0.3$ und $T = 1$ ergaben sich die in Abbildung 4.7 dargestellten Resultate; dabei wurden *nur* von den "linken" 10 Werkstücken n11, n13, n15, m5, ..., n21 die ersten gefundenen Parse (bzw. die daraus erzeugten Trainingsinstanzen) dem Lernmodul vorgelegt.

Die Ergebnisse zeigen, daß die Anzahl der erzeugten Patche für alle, also auch für die nicht angelernten Werkstücke reduziert werden konnte. Das heißt, daß eine Verallgemeinerung der Trainingsdaten in einem gewissem Maß


Anzahl
der
Patche
1000
800
600
400
200
Breitensuche
Heuristische Suche

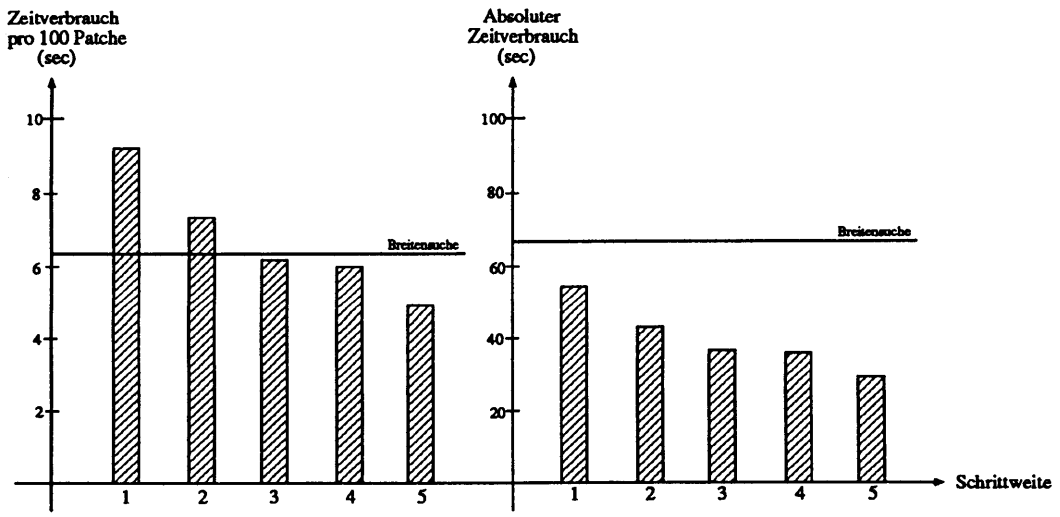


Abbildung 4.8: Der relative und absolute Zeitverbrauch bis zum Finden des ersten Parse für heuristische Suche beim 'Multispanblock'

weiten (Parameter und Netzarchitektur wie in Experiment 1). Überraschend ist, daß der relative Zeitverbrauch für größere Schrittweiten mit Heuristik *geringer* ist als bei Breitensuche (also ohne Heuristik), da ja für die Auswertung der Bewertungsfunktion *zusätzlicher* Berechnungsaufwand notwendig ist. Die Erklärung dieses Effekts liegt darin, daß mit der Reduktion der erzeugten Patche auch der Aufwand zur Kombination eines aus der Agenda ausgewählten Patches mit anderen Patchen verringert wird, da es dann weniger Kombinationsmöglichkeiten gibt. Im rechten Histogramm sieht man, daß mit der Reduktion der erzeugten Patche (vgl. Experiment 1) auch der Zeitverbrauch gegenüber Breitensuche verringert werden konnte, was ja letztendlich das Ziel einer heuristischen Steuerung ist. Die Abnahme des absoluten Zeitverbrauchs mit wachsender Schrittweite zeigt, daß die Reduktion der erzeugten Patche bei den Schrittweiten 1,...,5 etwa gleich war und somit der bei wachsender Schrittweite abnehmende relative Zeitverbrauch zum Tragen kam.

Abschließend sei nochmals betont, daß die durchgeführten Experimente lediglich *Provisorien* sind. Die Durchführung umfangreicher Testreihen hätte den Rahmen dieser Arbeit gesprengt.

Kapitel 5

Zusammenfassung und Ausblick

Die schlechte Komplexität von Graph-Parsern für allgemeine Graphgrammatiken legt die Verwendung von Heuristiken nahe. Aufgabe dieser Arbeit war es, zur Unterstützung des NRCFGG-Parsers GraPaKL eine Heuristikkomponente, die das benötigte Kontrollwissen durch ein Lernverfahren akquiriert, zu entwickeln.

Dazu wurden zunächst verschiedene Arten von Kontrollwissen diskutiert. Es ergab sich, daß die Repräsentation des Kontrollwissens in Form einer *Bewertungsfunktion* am sinnvollsten ist, weil das benötigte Kontrollwissen undurchschaubar und unscharf ist. Anschließend wurden verschiedene Lernverfahren, die sich zur Akquisition einer Bewertungsfunktion eignen, betrachtet. Es wurde festgestellt, daß die Verwendung eines *konnektionistischen* Lernverfahrens (Backpropagation) am günstigsten ist, da es zum einen "beliebige" Funktionen lernen kann und zum andern durch die verteilte Repräsentation eine besondere Fähigkeit zur Verallgemeinerung und Entdeckung von Korrelationen und Irrelevanzen besitzt.

Die entwickelte Heuristikkomponente besteht aus zwei Modulen, dem *Bewertungs-* und dem *Lernmodul*.

Das Bewertungsmodul steuert den Parser, indem es mittels einer Bewertungsfunktion Prioritäten an die Patche in der Agenda vergibt. Aufgrund der Tatsache, daß die Güte der Patche wesentlich vom Zustand des Parsers abhängt, setzt sich die Bewertungsfunktion aus zwei Teilen zusammen: einem *statischen*, d.h. vom aktuellen Zustand des Parsers unabhängigen, als auch einem *dynamischen*, also vom Parserzustand abhängigen Teil. Dabei kommt der dynamischen, situationsabhängigen Teilbewertung die Rolle der Primärsteuerung zu.

Die dynamische Teilbewertungsfunktion wird durch das Lernmodul ak-

quiriert. Der Benutzer bzw. Experte präsentiert Beispielparse, mit denen das Lernmodul die Bewertungsfunktion entsprechend verändert. Der Benutzer kann somit GraPaKL nur *mittelbar*, und zwar über das Lernmodul (durch die Präsentation von Beispielparsen) steuern; die konkrete Bewertungsfunktion bleibt dem Benutzer verborgen.

Wichtig ist, daß bei dieser Konzeption die Vollständigkeit des Parsers für "sinnvolle" Grammatiken (vgl. Abschnitt 2.2.3) erhalten bleibt; Patche werden weder aus dem Chart gelöscht noch von der Auswahl aus der Agenda ausgeschlossen, sondern allenfalls durch eine niedrige Bewertung zurückgestellt. Es können also bei Bedarf auf jedem Fall alle möglichen Parse erzeugt werden. Fraglich ist hingegen, ob sich diese heuristische Steuerung für beliebige Featuregrammatiken fair verhält; für sinnvolle Featuregrammatiken ist die Fairneß auf jeden Fall gegeben (siehe Abschnitt 2.2.3).

Bemerkenswert ist außerdem, daß die Architektur der Heuristikkomponente *unabhängig* von der zugrundeliegenden Graphgrammatik (Domäne) ist. Darüberhinaus läßt sie sich durchaus auch auf agenda-basierte Chart-Parser für Stringgrammatiken übertragen.

In (provisorischen) experimentellen Untersuchungen konnte die *prinzipielle* Eignung einerseits der Konzeption der Bewertungsfunktion und andererseits der Wahl des Lernverfahrens aufgezeigt werden. Ein systematisches Training mit verschiedenen Netzarchitekturen und Kombinationen der Parameter konnte im Rahmen dieser Arbeit nicht durchgeführt werden, da dies aufwendige Testreihen erfordert. Die Existenz *optimaler* Parsläufe zur Gewinnung von Trainingsdaten würde das Training wesentlich beschleunigen. Dies wirft die Frage auf, ob sich solche Parsläufe *automatisch* erzeugen lassen.

Offen ist außerdem, ob sich der Lernvorgang vollständig automatisieren läßt, so daß die Bewertungsfunktion ständig angepaßt wird während GraPaKL im Einsatz ist, ohne daß der Benutzer Beispielparse vorlegen muß. Dazu müßte das Lernmodul entweder Wissen über die Güte gefundener Parse haben oder mit einem anderen Expertensystem gekoppelt sein, damit es "gute" von "schlechten" Parsen unterscheiden kann. Ersteres kann wiederrum als Lerngegenstand betrachtet werden.

Schließlich ist zu sagen, daß eine Abschätzung der Komplexität der mit der Heuristik zusätzlich benötigten Berechnungen ebenfalls offen gelassen wurde. Diese hängt im wesentlichen von der Größe des Backpropagationnetzes und der (durchschnittlichen) Anzahl der in der Agenda befindlichen Patche ab.

Anhang A

Benutzung der Heuristikkomponente

Das GraPaKL-Handbuch findet sich in [17]. Im folgenden werden die für die Benutzung der Heuristikkomponente zusätzlich benötigten COMMON LISP Funktionen beschrieben.

Im Fall von heuristischer Suche muß ein neuronales Netz geladen sein. Bei der Kompilation einer Grammatik wird automatisch das Netz mit dem Namen `Grammatikname.net`[1] geladen (d.h. an die globale Variable `*net*` gebunden). Soll ein anderes Netz geladen werden, so ist dies explizit mit der entsprechenden Funktion (s.v.) nachzuladen.

Funktionen zum Einstellen von Optionen

`(strategy-opt {<strategy>})`[2]

Setzt die Suchstrategie des Parsers auf das *optionale* Argument `<strategy>`. Mögliche Werte für `<strategy>` sind:

`'breadth-first`	Breitensuche
`'depth-first`	Tiefensuche
`'heuristic`	Heuristische Suche

Der Defaultwert ist `heuristic`.

[1] Zu jeder Grammatik sollte also stets das Netz, welches bei der heuristischen Suche mit dieser Grammatik benutzt werden soll, mit `Grammatikname.net` bezeichnet und vorhanden sein.

[2] Die Syntax {<arg>} bezeichnet ein *optionales* Argument.

`(step-opt {<number>})`

Setzt die Schrittweite, also die Anzahl von Zustandsänderungen des Parsers, nach denen die dynamische Bewertung erneut aufgerufen wird, auf das *optionale* Argument `<number>`. Der Defaultwert ist 5.

`(example-opt {<truth-value>})`

Setzt die Option, ob in einem Parslauf gefundene und vom Benutzer (auf Anfrage des Systems) für "gut" befundene Parse zur Erzeugung von Trainingsinstanzen benutzt werden sollen, auf das *optionale* Argument `<truth-value>`. Der Defaultwert ist `nil`.

Ist die Option gesetzt, so wird der Benutzer beim Starten des Parsers nach einem Namen für die Datei gefragt, in der die erzeugten Trainingsinstanzen abgelegt werden. Existiert diese Datei noch nicht, so wird sie angelegt, ansonsten werden die Trainingsdaten in die schon existierende Datei geschrieben.

Funktionen zum Erzeugen, Speichern und Laden eines Backpropagationnetzes

`(make-net <hidden> <distribution> {<temp>})`

Erzeugt ein Backpropagationnetz mit genau einer verborgenen Schicht der Stärke `<hidden>`, welches der übersetzten Grammatik angepaßt ist (es muß also eine Grammatik übersetzt sein!) und bindet es an die globale Variable `*net*`. Das Ausmaß der Verteilung bei der Thermometerkodierung des Par-

angegeben. Das *optionale* Argument `<temp>` gibt den Temperaturparameter T der Aktivierungsfunktion an. Wird es weggelassen, so wird $T = 1$ gesetzt.

Die Anzahl der Eingabeneuronen des Netzes entspricht dem Produkt aus der Anzahl der Sorten (die in der Agenda auftreten können) und `<distribution>`. Die Anzahl der Ausgabeneuronen ist gleich der Anzahl der Sorten (die in der Agenda auftreten können).

`(save-net <filename>)`

Schreibt das Netz, das an `*net*` gebunden ist, in die Datei mit dem Namen `<filename>.net`. Das Argument `<filename>` muß ein String sein. Existiert

`(load-net <filename>)`

Lädt das Netz in der Datei mit dem Namen `<filename>.net` und bindet es an die globale Variable `*net*`. Das Argument muß ein String sein.

Die Lernfunktion

Bei Aufruf der Lernfunktion muß die entsprechende Grammatik kompiliert sein.

`(learn <file-list> <`η`> <`α`> {<times>})`

Die Lernfunktion trainiert das an `*net*` gebundene Netz. Dabei gibt `<file-list>` die Liste der Dateien (Strings!) mit den Trainingsinstanzen an, aus denen gelernt werden soll (diese Dateien können beim Parsen mit der durch den Aufruf `(example-opt t)` *eingeschalteten* Option angelegt werden). <η> gibt die Lernrate und <α> den Einfluß des Momentum-Terms an. Das *optionale* Argument `<times>` gibt die Anzahl der durchzuführenden Epochen an. Wird es weggelassen, so wird genau eine Epoche durchgeführt.

Nach Terminierung des Aufrufs der Lernfunktion sind die Gewichte des geladenen Netzes *modifiziert*.

Bsp.: `(learn '(''data1'' ''data2'') 1 0.3 4)`

Funktion zur Erstellung einer Statistik von Parsläufen

`(statistic <strategy> <grammar-list> <filename>)`

Parst alle verfügbaren Werkstücke der als `<grammar-list>` angegebenen Liste von Grammatiken (Strings!) und schreibt die statistischen Daten (Zeitverbrauch, Anzahl der erzeugten Patche) in die Datei mit dem Namen `<filename>`. Existiert diese Datei noch nicht, wird sie angelegt, ansonsten überschrieben.

Bsp.: `(statistic 'heuristic '(''milling'' ''design'') ''stat1'')`

Anhang B

Implementierung

Die Beschreibung der GraPaKL-Implementierung[1] findet sich in [17]. Im folgenden wird ein kurzer Überblick über die Integration der Heuristikkomponente in GraPaKL gegeben werden. Im wesentlichen wurden zwei bestehenden Module (`vars` und `pars`) modifiziert und erweitert sowie zwei neue Module (`eval` und `lear`) hinzugefügt.

Erweiterung des Moduls vars

Das Modul `vars`, welches die globalen Variablen und Konstanten enthält, wurde im wesentlichen um folgende Variablen erweitert:

`*heur-agenda-cp*`	Teil der strukturierten Agenda mit den kompletten Patchen
`*heur-agenda-pp*`	Teil der strukturierten Agenda mit den partiellen Patchen
`*parser-state*`	Zustand des Parsers
`*sort-priority-vector*`	Vektor mit den durch die dynamische Bewertung berechneten aktuellen Sortenprioritäten
`*net*`	Geladenes Backpropagationnetz
`*step-number*`	Schrittweite, also die Anzahl der Zustandsänderungen, nach denen die dynamische Bewertung erneut aufgerufen wird
`*search-history*`	Liste der während des Parslaufs getroffenen Auswahlentscheidungen
`*degree-of-distribution*`	Ausmaß der Verteilung bei der Thermometerkodierung

[1] GraPaKL ist in COMMON LISP implementiert.

Die Sorten werden über eindeutige Nummern referenziert, die den Sorten bei der Kompilation der Grammatik zugeordnet werden und in der Assoziationsliste `*agenda-sorts-with-cnt*` unter der jeweiligen Sorte stehen.

Erweiterung des Moduls pars

Das Modul `pars`, welches die Funktionen für den Parsing-Algorithmus enthält, wurde im wesentlichen um folgende Funktionen erweitert:

`heur-graph-parse`	Heuristische Version des Parsing-Algorithmus
`best-from-heur-agenda`	Auswahl eines besten Patch aus der strukturierten Agenda gemäß des aktuellen Inhaltes von `*sort-priority-vector*`
`add-to-heur-agenda`	Einfügen eines statisch bewerteten Patches in die strukturierte Agenda
`statistic`	Schreibt Zeit- und Patchstatistiken von Parsläufen in eine Datei

Das Modul eval

Das Modul `eval` enthält die Funktionen zur Steuerung von GraPaKL mit der Heuristikkomponente und zur Verwaltung der neuronalen Netze. Dies sind im wesentlichen

`static-evaluation`	Statische Bewertungsfunktion
`dynamic-evaluation`	Dynamische Bewertungsfunktion, die ihren Wert in `*sort-priority-vector*` speichert
`code-parser-state`	Thermometerkodierung des Parserzustands für die Netzeingabe
`net-out`	Berechnet die Netzausgabe
`make-net`	Erzeugt ein grammatikspezifisches Backpropagationnetz mit genau einer verborgenen Schicht
`save-net`	Schreibt ein Netz in eine Datei
`load-net`	Liest ein Netz aus einer Datei

Das Modul `lear`

Das Modul `lear` enthält die Funktionen für den Lernalgorithmus. Dies sind im wesentlichen

`learn`	Die Lernfunktion (lernt aus einer Menge von Trainingsinstanzen)
`learn-from-training-instance`	Der eigentliche Backpropagation-Algorithmus (lernt aus einer einzigen Trainingsinstanz)

Literaturverzeichnis

[1] K.-D. Althoff, S. Weß, B. Bartsch-Spörl, D. Janetzko, F. Maurer, A. Voß. Fallbasiertes Schließen in Expertensystemen: Welche Rolle spielen Fälle für wissensbasierte Systeme ? In: Künstliche Intelligenz 4/92 (Zeitschrift des Fachbereichs 1 der Gesllschaft für Informatik), 1992.

[2] A. Bernardi, C. Klauck, R. Legleitner. FEAT-REP: Representing Features in CAD/CAM. 4th International Symposium on Artificial Intelligence: Applications in Informatics, 1991.

[3] K. Bläsius, N. Eisinger, J. Siekmann, G. Smolka, A.Herold, C. Walther. The MARKGRAF KARL Refutation Procedure. 7th International Joint Conference on Artificial Intelligence, pp. 511-518, 1981.

[4] S. Bradtke, W.G. Lehnert. Some Experiments with Case-Based Search. 7th National Conference of the American Association on Artificial Intelligence (AAAI), pp. 133-138, 1988.

[5] R. Davis. Meta-Rules: Reasoning about Control. Artificial Intelligence 15, pp. 179-222, 1980.

[6] N.R. Draper, H. Smith. Applied regression analysis (2nd Edition). Wiley Series in Probability and Mathematical Statistics, 1980.

[7] R.O. Duda, P.E. Hart. Pattern Classification and Scene Analysis. Wiley Interscience Publication, 1973.

[8] D.E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[9] R. Hannuschka, U. Hartmann, M. Mandischer, V. Weber. Steuerung von logischen Beweisen mit konnektionistischen Modellen. In Kopplung deklarativer und konnektionistischer Wissensrepräsentation. Endbericht der Projektgruppe PANDA, Universität Dortmund, Berichtnummer 352, pp. 37-89, 1990.

[10] J.H. Holland. Adaption in Natural and Artificial Systems. University of Michigan Press, 1975.

[11] T. Kawakami, M. Minagawa, Y. Kakazu. Auto Tuning of 3-D Packing Rules Using Genetic Algorithms. IEEE/RSJ International Workshop on Intelligent Robots and Systems (IROS), pp. 1319-1324, 1991.

[12] C. Klauck. Eine Graphgrammatik zur Feature-Repräsentation und Feature-Erkennung in CAD/CAM. Dissertation, Universität Kaiserslautern, 1993.

[13] J.E. Laird, A. Newell, P.S. Rosenbloom. SOAR: An Architecture for General Intelligence. Artificial Intelligence 33(1), pp. 1-64, 1987.

[14] P. Langley. Learning to Search: From Weak Methods to Domain-Specific Heuristics. Cognitive Science 9, pp. 217-260, 1985.

[15] K.-F. Lee, S. Mahajan. A Pattern Classification Approach to Evaluation Function Learning. Artificial Intelligence 36, pp. 1-25, 1988.

[16] R. Levinson, R. Snyder. Adaptive Pattern-Oriented Chess. 9th National Conference of the American Association on Artificial Intelligence (AAAI), pp. 601-606, 1991.

[17] J. Mauss. Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken. Document D-92-10, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1992.

[23] E. Rich, K. Knight. Artificial Intelligence (2nd Edition). McGraw-Hill, 1991.

[24] M.M. Richter. Prinzipien der Künstlichen Intelligenz. B.G. Teubner, 1989.

[25] D.E. Rumelhart, G.E. Hinton, R.J. Williams. Learning Internal Representations by Error Propagation. In: Parallel Distributed Processing Vol. 1,2 und 3, MIT-Press, 1986.

[26] A.L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal, pp. 211-229, Juli 1959.

[27] J.R. Slagle, C.D. Farrell. Experiments in Automatic Learning for a Multipurpose Heuristic Program. Communications of the ACM 14(2), pp. 91-99, 1971.

[28] C. Suttner, W. Ertel. Automatic Acquisition of Search Guiding Heuristics. 10th International Conference on Automated Deduction (CADE), pp. 470-484, 1989.

[29] R.S. Sutton. Learning to Predict by the Method of Temporal Differences. Machine Learning 3, pp. 9-43, 1988.

[30] P.E. Utgoff, S. Saxena. Learning a Preference Predicate. 4th International Workshop on Machine Learning, pp. 115-121, 1987.

[31] R.E. Walpole, R.H. Myers. Probability and Statistics for Engineers and Scientists (3rd Edition). McMillan Publishing, 1985.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
67608 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi: Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-53
Werner Stephan, Susanne Biundo:
A New Logical Framework for Deductive Planning
15 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
13 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
19 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist: Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi: An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-04
Christoph Klauck, Johannes Schwagereit: GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz: Satisfiability of the Smallest Binary Program
8 Seiten

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Juergen Buerckert: Reasoning about Temporal Relations: A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta: A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen: Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-17
Rolf Backofen: Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder: Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
Manfred Meyer, Jörg Müller: Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy: Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel: Document Highlighting — Message Classification in Printed Business Letters
17 pages

RR-93-25
Klaus Fischer, Norbert Kuhn: A DAI Approach to Modeling the Transportation Domain
93 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger: Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

RR-93-29
Armin Laux: Representing Belief in Multi-Agent Worlds viaTerminological Logics
35 pages

DFKI Technical Memos

TM-91-15
Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Liiuan Zhang: Entwurf und Implementierung eines

DFKI Documents

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.):
Kooperierende Agenten
78 Seiten

D-92-25

D-93-05
Elisabeth André, Winfried Graf, Jochen Heinsohn, Bernhard Nebel, Hans-Jürgen Profitlich, Thomas Rist, Wolfgang Wahlster:
PPP: Personalized Plan-Based Presenter
70 pages

D-93-06
Jürgen Müller (Hrsg.):
Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-07
Klaus-Peter Gores, Rainer Bleisinger:
Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch: Ein Generator

Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf eine Kombination zur Unterstützung eines Chart-Parsers

Robert Laux

D-93-15
Document