HEMERSON PISTORI

TECNOLOGIA ADAPTATIVA EM ENGENHARIA DE COMPUTAÇÃO: ESTADO DA ARTE E APLICAÇÕES

Texto apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Engenharia Elétrica.

São Paulo
2003

HEMERSON PISTORI

TECNOLOGIA ADAPTATIVA EM ENGENHARIA DE COMPUTAÇÃO: ESTADO DA ARTE E APLICAÇÕES

Texto apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Engenharia Elétrica.

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. João José Neto

São Paulo
2003

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ... de .... de 200...

Assinatura do autor

Assinatura do orientador

Ficha Catalográfica

PISTORI, HEMERSON

TECNOLOGIA ADAPTATIVA EM ENGENHARIA DE COMPUTAÇÃO: ESTADO DA ARTE E APLICAÇÕES. EDIÇÃO REVISADA. São Paulo, 2003. 174 p.

Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Dispositivos Adaptativos. 2. Aprendizagem de Máquina. 3. Engenharia de Computação. I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais. II. Título.

à Melissa, Julia, Hemerson, Emília, Alcides, Tatiane, Anderson e Jeferson.

AGRADECIMENTOS

Agradeço à minha esposa e filhos, Melissa, Hemerson e Julia pela paciência, compreensão e estímulo durante estes anos de idas e vindas entre Campo Grande e São Paulo. Aos meus pais e irmãos, Emília, Alcides, Anderson, Jeferson e Tatiane Raquel pelo carinho, companheirismo, churrascos ...

Ao meu orientador e amigo João José Neto cuja influência em minha vida certamente não se limitou ao campo técnico-científico.

Aos amigos Mauro e Cintia pela amável acolhida no meu primeiro ano em São Paulo.

Aos amigos Leonardo, Fábio, Rafael, Said e Regina pelo convívio durante meu segundo ano em São Paulo.

Aos novos amigos Eduardo e Luiz Fernando pelas agradáveis conversas.

Aos amigos e professores do curso de Engenharia de Computação da UCDB: Marco, Amaury, Conceição, Leonardo, Eugene, Alfredo, Luciano, Ivanilde, Maria Helena e Orlando.

Aos professores Paulo Muniz e Ricardo Rocha, que participaram da minha banca de qualificação, pelas revisões e comentários valiosos.

À Universidade Católica Dom Bosco pela confiança em mim depositada ao proporcionar as condições materiais e profissionais que possibilitaram a minha capacitação científica.

RESUMO

Neste trabalho é apresentado um conjunto de contribuições teóricas e práticas que buscam solidificar alguns conceitos da teoria dos dispositivos adaptativos baseados em regras, enfatizando a sua alta aplicabilidade. Uma ferramenta de apoio ao desenvolvimento de autômatos adaptativos, incluindo recursos de animação gráfica, foi desenvolvida de acordo com uma nova proposta de formalização que deverá complementar e simplificar a proposta original. A principal complementação está relacionada com a interpretação e a implementação de funções adaptativas, em sua forma mais geral: com ações elementares de consulta podendo retornar resultados múltiplos. A nossa proposta de formalização, que inclui um algoritmo para a execução de funções adaptativas, é uma ferramenta importante na determinação do impacto da execução da camada adaptativa no cálculo de complexidade geral de um autômato adaptativo. A tese apresenta também uma técnica para a integração de dispositivos adaptativos, basicamente discretos, com mecanismos capazes de manipular informação não-discreta. É mostrado também como estes resultados teóricos e as ferramentas desenvolvidas podem ser aplicadas na solução de problemas nas áreas de aprendizagem computacional, construção de compiladores, interface homem-máquina, visão computacional e diagnóstico médico.

ABSTRACT

This work presents a practical and theoretical assembly of contributions that consolidates some concepts from the rule-driven adaptive devices theory, emphasizing their high applicability. A supporting tool for the development of adaptive automata, which includes graphical animation resources, has been implemented, in agreement with our proposal of formalization. This proposal aims to complement and simplify the original proposal by including an in-depth analysis and formalization of adaptive functions implementation, in their most general form: with elementary query actions being able to return multiple results. The new formalization of adaptive functions, which includes an algorithm for adaptive function execution, is an important tool for determining the impact of an adaptive layer on the complexity analysis of general adaptive automata. The thesis also presents a new technique for the integration of adaptive automata with mechanisms for the manipulation of continuous values. Finally, the application of these theoretical results and the tools developed, to the solution of problems in the area of machine learning, compiler construction, man-machine interface, computational vision and medical diagnosis, is demonstrated.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

1 INTRODUÇÃO . . . **1**

1.1 Justificativa . . . 2

1.2 Objetivos . . . 3

1.3 Organização do Texto desta Tese . . . 5

I Revisão Bibliográfica . . . **6**

2 TECNOLOGIA ADAPTATIVA . . . **7**

2.1 Formalismos Adaptativos no Contexto Mundial . . . 14

2.2 Dispositivos Guiados por Regras Adaptativos . . . 15

2.2.1 Autômatos Adaptativos . . . 18

3 APRENDIZAGEM DE MÁQUINA . . . **23**

3.1 Aprendizagem de Máquina e Árvores de Decisão . . . 25

4 CÁLCULO DE PREDICADOS E UNIFICAÇÃO . . . **28**

4.1 Unificação . . . 34

4.2 Satisfação Seqüencial de Restrições . . . 37

II Resultados Teóricos **39**

5 ÁRVORES DE DECISÃO ADAPTATIVAS **40**

5.1 Árvores de Decisão Não-Determinísticas 40

5.1.1 Exemplo de Árvore de Decisão Não-Determinística 44

5.2 Árvores-DND Adaptativas 45

5.2.1 Exemplo ilustrativo 1 47

5.2.2 Exemplo Ilustrativo 2 48

5.3 AdapTree-E 49

5.3.1 Exemplo Ilustrativo 3 51

5.3.2 Tratamento de valores contínuos, ausentes e inconsistentes . . 52

5.3.3 Análise de Complexidade no Tempo e no Espaço 56

5.3.4 Experimentos 59

5.4 Consideração sobre Ordenação dos Atributos 62

5.5 Conclusões 64

6 FUNÇÕES ADAPTATIVAS **66**

6.1 Modelagem de Ações Adaptativas Elementares 66

6.2 Funcionamento do Algoritmo de Execução de Função Adaptativa . . . 73

6.3 Considerações sobre a ordem de aplicação das ações elementares de consulta 76

6.4 Complexidade do Algoritmo de Execução de Função Adaptativa . . . 77

6.5 Considerações sobre Cálculo de Complexidade de Dispositivos Adaptativos 79

6.6 Conclusão 82

7 AUTÔMATOS DE ESTADOS FINITOS ADAPTATIVOS **84**

7.1 Formalização 85

7.2 Um $\mathscr{A}$-AEF que reconhece uma linguagem que não é livre de contexto 87
7.3 Aprendizagem por memorização utilizando um $\mathscr{A}$-AEF 87
7.4 Conclusão 89

III Ferramentas 91

8 ADAPTOOLS 92

8.1 Autômatos Adaptativos no AdapTools 92
8.1.1 Diferenças em Relação à Definição Original 96
8.1.2 Formato do Arquivo Gerado pelo AdapTools 97
8.2 Interface do Sistema 97
8.2.1 Visualização Gráfica 99
8.3 Tratamento de Rotinas Semânticas 100
8.4 Comunicação entre Máquinas Virtuais 100
8.5 Integração com Outros Sistemas 101
8.6 Arquitetura do Sistema 101
8.7 Considerações sobre a Máquina Virtual do AdapTools 102
8.8 Conclusões 103

9 INTEGRAÇÃO DE DISPOSITIVOS ADAPTATIVOS, DE APRENDIZAGEM DE MÁQUINA E DE VISÃO COMPUTACIONAL 105

9.1 Descrição dos Pacotes Utilizados 106
9.1.1 Processamento Digital de Imagens 106
9.1.2 Aprendizagem de Máquina 107
9.1.3 Tratamento de Dispositivos de Captura de Imagens 107
9.2 Desenvolvimento do Pacote Integrado 107
9.3 Conclusões 109

IV Aplicações 110

10 PROCESSAMENTO DIGITAL DE IMAGENS E APRENDIZAGEM DE MÁQUINA 111

10.1 Interação Homem-Máquina através de Línguas de Sinais 111

10.1.1 Trabalhos Relacionados 113

10.1.2 Língua de Sinais Brasileira 115

10.1.3 Desenvolvimento 117

10.1.4 Experimentos 120

10.1.5 Conclusão 120

10.2 Interação Homem-Máquina através do Olhar 121

10.2.1 Introdução 121

10.2.2 Técnicas de Detecção da Direção do Olhar 123

10.2.3 Desenvolvimento 125

10.2.4 Experimentos 128

10.2.5 Conclusões 128

11 COMPILADORES E FERRAMENTAS EDUCACIONAIS 131

11.1 Meta-reconhecedor Wirth para AdapTools 131

11.2 Recuperação Automática de Erros 133

11.2.1 Recuperação de Erros Simples - Método Clássico 134

11.2.2 Recuperação de Erros Adaptativa 136

11.2.3 Implementação no AdapTools 137

11.2.4 Conclusões 137

11.3 Tradução Texto-Voz 138

12 INTELIGÊNCIA ARTIFICIAL 140

12.1 Diagnóstico Médico 140
12.2 Ambiente Interativo para Manipulação de Árvores de Decisão 141

13 CONCLUSÕES 144

13.1 Contribuições 144
13.2 Sugestões para Trabalhos Futuros 145

Anexos 147

A Autômatos de Pilha Estruturados 147

B Código AdapTools para Aprendizagem por Memorização 149

C Tratamento de Rotinas Semânticas 150

D Clonador 156

E Meta-Compilador Wirth para AdapTools 157

F Um Exemplo de Recuperação de Erros usando Autômatos Adaptativos 160

G Implementação da Semântica do Tradutor Texto-Voz 161

Referências Bibliográficas 163

LISTA DE FIGURAS

2.1 Principais acontecimentos nacionais na história dos dispositivos adaptativos 13

2.2 Linha de Tempo para Formalismos Adaptativos 16

3.1 Duas árvores de decisão representando a mesma função f 25

4.1 Gramática livre de contexto para fórmulas bem formadas do cálculo de predicados 32

5.1 Exemplo de uma árvore de decisão 41

5.2 Gramática para funções adaptativas em árvores-DND adaptativas 46

5.3 Evolução de uma AdapTree 51

5.4 Árvore-DND classificando exemplo com valores ausentes 59

5.5 Impacto da ordem dos atributos no desempenho de uma AdapTree-E . 63

6.1 Exemplo de um autômato de estados finitos 67

6.2 Autômato após a execução das ações elementares 69

6.3 Diferentes interpretação para geradores 70

6.4 Autômato adaptativo antes da execução do algoritmo 6.1 75

6.5 Árvore de chamadas recursivas do algoritmo 6.2 76

6.6 Autômato subjacente após a execução do algoritmo 6.1 76

6.7 Árvore de chamadas recursivas com outra ordenação para consultas . 77

6.8 Autômato adaptativo com complexidade exponencial no tempo 80

7.1 Autômato de estados finitos adaptativo que reconhece $a^n b^n c^n$ 87

7.2 $\mathscr{A}$-AEF M que realiza aprendizagem por memorização (o estado q_f é propositadamente inalcançável) 88

7.3 Função Π para exemplo da aprendizagem por memorização 89
7.4 M Após a leitura do símbolo a 89
7.5 M Após a leitura da cadeia aM 89
8.1 Autômato adaptativo que reconhece $a^n b^n c^n$ 95
8.2 Simulador de pilhas na versão original 96
8.3 Simulador de pilha no AdapTools 96
8.4 Janela principal do AdapTools 98
8.5 Módulo de animação gráfica do AdapTools 99
9.1 Sistema guiado por sinais visuais 108
10.1 Exemplos de sinais icônicos em LIBRAS 115
10.2 Alguns símbolos do alfabeto LIBRAS 116
10.3 Pré-processamento no protótipo LIBRAS 118
10.4 Parâmetros extraídos de três imagens diferentes 119
10.5 Ambiente em que os experimentos com o vTTT foram realizados . . . 122
10.6 Detecção da região dos olhos no vTTT 126
11.1 Meta-reconhecedor Wirth para AdapTools 132
11.2 Exemplo da execução do algoritmo de recuperação de erros 136
11.3 Funções adaptativas para recuperação de erro 136
11.4 Algumas transições do autômato de tradução texto-voz 139
12.1 Tela principal do xILE 142

LISTA DE TABELAS

3.1 Função f para o exemplo de árvore de decisão 25

4.1 Exemplos aritméticos envolvendo objetos, funções e relações 29

4.2 Exemplos gerais envolvendo objetos, funções e relações 30

4.3 Exemplos de sentenças lógicas 30

4.4 Exemplos de sentenças quantificadas 31

4.5 Exemplos de sentenças na forma clausal 33

4.6 Exemplos de Substituições 35

4.7 Exemplo de base de conhecimento para SSR 37

5.1 Função adaptativa do exemplo 1 48

5.2 Funções adaptativas do exemplo 3 52

5.3 AdapTree-E com diferentes métodos de discretização 55

5.4 Experimentos sobre o custo em espaço do AdapTree-E 57

5.5 Taxas de acerto para AdapTree-E, naiveBayes, KNN, C4.5 e Id3 . . . 60

5.6 Descrição dos conjuntos de dados usados no experimento com AdapTree-E 61

5.7 Comparação de Tempo para Classificação 63

10.1 Resultados comparativos no protótipo LIBRAS 120

10.2 Matriz de convolução para detecção de bordas 126

10.3 Resultados comparativos no vTTT 129

LISTA DE ABREVIATURAS E SIGLAS

AD Árvore de Decisão

AdapTree-E *Adaptive Tree Extended*

AEF Autômato de Estados Finitos

AP Autômato de Pilha

APE Autômato de Pilha Estruturado

BNF *Backus-Naur Form*

IA Inteligência Artificial

IHM Interação Homem-Máquina

IUE Incontinência Urinária de Esforço

JMF *Java Media Framework*

k-NN *k-Nearest Neighbour*

LIBRAS Língua BRAsileira de Sinais

LTA Laboratório de linguagens e Tecnologias Adaptativas

MDL *Minimum Description Length*

NIH *National Institute of Health*

PCA *Principal Component Analysis*

PECC Percentual de Exemplos Corretamente Classificados

RGB *Red, Green and Blue*

SLD *Selection function, Linear resolution and Definite clauses*

SPD Sistema de apoio ao Projeto e desenvolvimento de sistemas Digitais

SSR Satisfação Seqüencial de Restrições

UCI *University of California-Irvine*

UMG Unificador Mais Geral

VMDS *Virtual Machine Data Structures*

vTTT *Vision based Tic-Tac-Toe*

WEKA *Waikato Environment for Knowledge Acquisition*

WWF *Well-Formed Formulas*

xILE *Incremental Learning Environment for X-Systems*

1 INTRODUÇÃO

O formalismo geral que caracteriza os dispositivos adaptativos foi apresentado pela primeira vez à comunidade científica em (NETO, 2001), embora, em formulações mais restritas, venha sendo utilizado há muito tempo (CHRISTIANSEN, 1986; CHRISTIANSEN; SHAW, 1990; BURSHTEYN, 1990b; CABASINO; PAOLUCCI; TODESCO, 1992; NETO, 1993; RUBINSTEIN; SHUTT, 1993; BOULLIER, 1994). Um dispositivo adaptativo é constituído de um mecanismo subjacente, como por exemplo, autômatos, gramáticas, árvores de decisão, etc., ao qual é acrescido o que se denomina mecanismo adaptativo, responsável por permitir que a estrutura do mecanismo subjacente seja dinamicamente modificada. Um autômato de estados finitos, por exemplo, quando acrescido de um mecanismo adaptativo, passa a poder sofrer remoções ou inserções de transições enquanto processa uma cadeia de entrada, o que aumenta sobremaneira sua capacidade de expressão. De fato, é provado que autômatos adaptativos possuem o mesmo poder de expressão das máquinas de Turing (ROCHA; NETO, 2000a).

Uma característica fundamental das tecnologias derivadas da teoria dos dispositivos adaptativos (tecnologias adaptativas) é a possibilidade de reaproveitamento integral de formalismos consolidados, com aumento do seu poder de representação, ao custo de um pequeno acréscimo na sua complexidade formal. As primeiras aplicações de tecnologias adaptativas concentraram-se na área de construção de compiladores (NETO, 1993), com a busca de mecanismos puramente sintáticos para resolver problemas que não podem ser tratados pelos limitados, mas extensivamente utilizados, autômatos a pilha. Basicamente, como os autômatos a pilha são capazes de reconhecer apenas linguagens livres de contexto e, como a quase totalidade das linguagens de programação de interesse não são livres de contexto, parte da análise, que poderia ser efetuada de maneira puramente sintática, acaba não podendo ser considerada na fase de análise sintática. As características da linguagem de programação intratáveis pelos métodos tradicionais na análise sintática acabam sendo efetuadas utilizando-se de outros recur-

sos, em uma fase geralmente denominada análise semântica estática. Com a tecnologia adaptativa é possível aumentar o poder de expressão do autômato a pilha preservando-se boa parte do seu formalismo. Com isto, abre-se a oportunidade de construção de geradores automáticos de *parsers* capazes de resolver, inclusive, as dependências de contexto presentes na linguagem fonte.

1.1 Justificativa

Assim como praticamente toda a tecnologia envolvida na construção de compiladores, as tecnologias adaptativas também já estão sendo utilizadas em outras áreas, como em processamento de linguagem natural, em que a capacidade de expressão dos formalismos computacionais são ainda mais importantes (NETO; MORAES, 2002; MENEZES; NETO, 2002; MENEZES, 2000). Na área de robótica e visão computacional, os dispositivos adaptativos estão sendo pesquisados para solucionar problemas de planejamento e navegação autônoma (JUNIOR; NETO; HIRAKAWA, 2000), e problemas de reconhecimento de padrões a partir de representações sintáticas de entes geométricos (COSTA; HIRAKAWA; NETO, 2002). Também existem dispositivos adaptativos aplicados à engenharia de software de sistemas de tempo real, cujos mecanismos subjacentes são *statecharts* (NETO; JUNIOR; SANTOS, 1998); e um interessante software de composição musical automática, baseado em redes de Markov adaptativas, que produz, em tempo real, melodias em estilo barroco (BASSETO; NETO, 1999). Um levantamento das principais contribuições da tecnologia adaptativa, desde sua introdução há cerca de 20 anos, pode ser encontrado no capítulo 2 desta tese.

O formalismo adaptativo mostra-se uma alternativa bastante natural para a modelagem de soluções na área de aprendizagem computacional, uma vez que é capaz de capturar um aspecto fundamental da aprendizagem: a adaptação dinâmica das estruturas internas de um mecanismo em função de sua interação com o ambiente. No entanto, poucos são os trabalhos que consideram esta questão de um maneira ampla, levando em conta problemas como o tratamento de valores contínuos, informação inconsistente e incompleta. Além disto, a plena difusão da tecnologia adaptativa esbarra em um outro problema, que é a baixa disponibilidade de ferramentas computacionais, tanto educacionais quanto de desenvolvimento, que facilitem sua compreensão e utilização. Talvez por isto, problemas específicos e detalhes que são apenas descobertos quando se implementa um autômato adaptativo, por exemplo, ainda não tenham

sido suficientemente explorados.

1.2 Objetivos

O primeiro objetivo deste trabalho foi o de analisar e propor uma forma de construção de algoritmos de aprendizagem de máquina utilizando conceitos oriundos da teoria dos dispositivos adaptativos. O resultado disto foi a criação de um mecanismo híbrido de aprendizagem, chamado AdapTree-E, envolvendo técnicas discretas e contínuas, que apresenta desempenho comparável ao das soluções clássicas existentes, além de oferecer uma alternativa inovadora para a modelagem de problemas de aprendizagem da máquina. Experimentos simulados e reais foram efetuados sobre problemas em diferentes áreas, para que tanto a abrangência quanto a eficácia do método proposto pudessem ser analisadas. No desenvolvimento desses experimentos produzimos uma ferramenta interessante que facilita a construção de sistemas cuja interface é guiada por sinais visuais, capturados em tempo real por uma filmadora.

O desenvolvimento de ferramentas que facilitam a reutilização das idéias discutidas neste trabalho e um maior aprofundamento nos problemas relacionados com implementação de tecnologia adaptativa foi o segundo objetivo principal do trabalho. Implementamos um ambiente de desenvolvimento para autômatos adaptativos, o AdapTools, que serve também como uma ferramenta educacional para o ensino da tecnologia adaptativa. O ambiente conta com recursos de edição, depuração, animação gráfica, controle de projetos, tutorais e exemplos. Os códigos-fonte do software são abertos, documentados e escritos em Java, principalmente por ser esta uma linguagem altamente portável.

Para implementar as ferramentas acima citadas tivemos que nos aprofundar bastante na teoria dos autômatos adaptativos. Este aprofundamento nos permitiu visualizar e propor alternativas para algumas das questões relacionadas com a definição original. Esta tese desenvolve um embasamento teórico complementar para o funcionamento das ações adaptativas elementares, principalmente em relação às consultas que retornam mais de uma transição. Esta teoria foi construída sobre conceitos sólidos do cálculo de predicados e da unificação. São introduzidas adicionalmente algumas simplificações na definição original, buscando facilitar sua utilização, sem perdas significativas na expressividade do modelo resultante. Esta nova formalização permitiu

a realização de análises de complexidade em tempo e espaço bem mais detalhadas e precisas que as existentes.

O quarto e último objetivo desta tese foi demonstrar a alta aplicabilidade da tecnologia adaptativa na área da engenharia de computação. Utilizando uma implementação do mecanismo AdapTree-E e a ferramenta AdapTools foi possível projetar e implementar uma série de dispositivos visando a solução de diversos problemas reais nas áreas de construção de compiladores, tradução texto-voz, interação homem-máquina, visão computacional, educação e inteligência artificial. A aplicação destas ferramentas na solução de uma variada gama de problemas, além dos diversos ensaios e testes realizados durante o desenvolvimento, nos permitiu também atingir um bom domínio sobre técnicas de solução de problemas utilizando dispositivos adaptativos.

Segue abaixo um resumo dos resultados teóricos obtidos como produtos da elaboração desta tese:

- Complementação da formalização das funções adaptativas, incluindo o tratamento de ações elementares de consulta envolvendo variáveis referenciadas em diferentes ações e com múltiplos resultados.
- Um arcabouço teórico que facilita a elaboração de cálculos de complexidade e a análise da expressividade das funções adaptativas.
- Uma nova técnica para a integração do formalismo adaptativo, que trabalha basicamente com informação discreta, com formalismos que permitam a manipulação de valores contínuos.
- Uma revisão e simplificação do formalismo original que define os autômatos adaptativos, incluindo melhorias na formalização, na notação e na metodologia de utilização dos modelos.

Também alcançamos os seguintes resultados práticos nesta pesquisa:

- Um ambiente completo, confortável e operante, para desenvolvimento, depuração, teste e simulação de autômatos adaptativos.
- Implementação de diversos autômatos adaptativos importantes, utilizando o ambiente construído.

- Aplicações de dispositivos adaptativos nas áreas de visão computacional, construção de compiladores, tradução texto-voz, inteligência artificial e aprendizagem de máquina.

1.3 Organização do Texto desta Tese

Esta tese está divida em quatro grandes partes: revisão bibliográfica, resultados teóricos, ferramentas e aplicações. Os três capítulos da primeira parte oferecem uma revisão da literatura nas áreas de dispositivos adaptativos (cap. 2), aprendizagem de máquina (cap. 3) e cálculo de predicados (cap. 4). No capítulo sobre cálculo de predicados abordaremos também o conceito de unificação, que junto com a satisfação seqüencial de restrições (uma especialização do cálculo de predicados), formam o núcleo conceitual para a formalização de funções adaptativas proposta nesta tese.

A parte dois, dedicada aos resultados teóricos da tese, é composta de um capítulo sobre árvores de decisão adaptativas e tratamento de valores contínuos (cap. 5) , outro sobre complementação da formalização de funções adaptativas (cap. 6) e um último sobre autômatos de estados finitos adaptativos (cap. 7) . As terceira e quarta partes da tese apresentam, respectivamente, as ferramentas desenvolvidas e uma série de aplicativos implementados com o auxílio dessas ferramentas. Foram duas as ferramentas desenvolvidas: o AdapTools (cap. 8) e um ambiente de apoio ao desenvolvimento de sistemas guiados por sinais visuais (cap. 9). As aplicações foram organizadas em três grandes grupos: processamento digital de sinais e aprendizagem de máquina (cap. 10); construção de compiladores e software educacional (cap. 11) e inteligência artificial (cap. 12). O último capítulo é dedicado às conclusões e sugestões para trabalhos futuros.

Parte I

Revisão Bibliográfica

2 TECNOLOGIA ADAPTATIVA

A teoria dos dispositivos adaptativos baseados em regras nasceu da busca por formalismos tão simples de usar quanto, por exemplo, os autômatos de estados finitos, mas capazes de representar problemas mais complexos, envolvendo linguagens não-regulares, e até mesmo dependentes de contexto. Uma das idéias centrais no formalismo adaptativo é que dispositivos mais poderosos, em relação à capacidade de expressão, podem ser obtidos a partir de uma progressão suave dos recursos oferecidos por um dispositivo mais simples. Neste sentido, os autômatos de pilha estruturados (NETO; MAGALHÃES, 1981a; NETO; PARIENTE; LEONARDI, 1999; NETO, 1987) podem ser considerados como precursores dos dispositivos adaptativos, uma vez que estendem o poder de autômatos de estados finitos, mantendo praticamente intacta a sintaxe original: autômatos de pilha estruturados são basicamente conjuntos de autômatos de estados finitos.

A possibilidade de construção de um reconhecedor sintático simples e poderoso, utilizando autômatos de pilha estruturados, foi demonstrada em (NETO; MAGALHÃES, 1981a), juntamente com um estudo comparativo que indica a equivalência expressiva entre autômatos de pilha estruturados (APE) e autômatos de pilha (AP). Os autômatos de pilha estruturados, juntamente com técnicas para testes de consistência, transformações gramaticais, geração de APEs a partir de gramáticas na notação BNF (*Backus-Naur Form*), remoção de não-determinísmos e geração de diagnósticos, foram utilizados na implementação de um sistema automático de apoio ao projeto e desenvolvimento de sistemas digitais, o SPD (NETO; MAGALHÃES, 1981b, 1983; NETO, 1983b). Este sistema foi provavelmente um dos primeiros, deste gênero, a ser desenvolvido inteiramente no Brasil, ainda na década de 70. A utilização do SPD como ferramenta para simulação de máquinas novas e/ou indisponíveis, incluindo a possibilidade de geração automática de "*cross-assemblers*", um avanço na área de projeto de sistemas digitais no Brasil, foi discutida em (NETO, 1982, 1983a).

Um outro trabalho importante que precede a introdução dos dispositivos adaptativos foi um estudo detalhado sobre o projeto de geradores autômaticos de reconhecedores sintáticos utilizando gramáticas descritas em notação de Wirth modificada (NETO, 1987), apresentado em (NETO, 1988a). A transição natural entre gramáticas na notação de Wirth modificada e autômatos de pilha estruturados foi explorada elegantemente em uma aplicação pedagógica em construção de compiladores, que demonstra como um gerador de *parsers* pode ser apresentado e assimilado de uma maneira bastante simples e intuitiva (NETO, 1987; NETO; PARIENTE; LEONARDI, 1999).

Autômatos de pilha estruturados, no entanto, apresentam um poder de expressão limitado pelas linguagens livres de contexto, o que impede sua utilização direta em uma diversidade de problemas. Seguindo o mesmo princípio de progressão suave de recursos oferecidos, foram criados os autômatos adaptativos (NETO, 1988b, 1993, 1994): autômatos de pilha estruturados que podem ter sua estrutura modificada durante sua operação. Autômatos adaptativos se destacam como uma interessante alternativa para as máquinas de Turing (NETO, 2000), tendo o mesmo poder de expressão que elas (ROCHA; NETO, 2000a), mas apresentando a desejável característica de poderem ser especificados como simples extensões dos autômatos de pilha estruturados.

O conceito de adaptabilidade introduzido com os autômatos adaptativos foi depois desatrelado de seu mecanismo subjacente original, o autômato de pilha estruturado, para ser reutilizado em um novo domínio: o da especificação e projeto de sistemas reativos complexos [1]. Neste domínio, um formalismo de ampla utilização é o *statechart*, principalmente por incorporar mecanismos de representação hierárquica. Uma versão adaptativa deste formalismo, o *statechart* adaptativo, resultou na implementação da primeira ferramenta computacional para edição e simulação de um dispositivo adaptativo (NETO; JUNIOR; SANTOS, 1998; JUNIOR, 1995; NETO; JUNIOR, 1999b, 1999a): o STAD - *STAtecharts* ADaptativos. Uma evolução do STAD, incorporando recursos de sincronização de processos, baseados em redes de Petri, foi implementada pouco tempo depois, no sistema SAS (ou STAD-S): *Statecharts* Adaptativos Sincronizados (SANTOS, 1997).

A primeira implementação de um ambiente integrado para desenvolvimento de autômatos adaptativos, ou de uma versão modificada deste formalismo, foi o RSW

[1] Sistemas, em grande parte, guiados por eventos, e que necessitam reagir continuamente a estímulos externos e internos, como por exemplo, sistemas operacionais de computadores, redes de telefonia e sistemas para controle de mísseis e aviões (HAREL, 1987)

(meta-Reconhecedor Sintático para Windows) (PEREIRA; NETO, 1997). O sistema RSW oferece um ambiente completo para edição, compilação e execução de "programas" escritos na linguagem RSW, uma linguagem bastante próxima da notação definida no trabalho original sobre autômatos adaptativos (NETO, 1993, 1994). Embora o formalismo dos autômatos adaptativos não tenha sido implementado na sua totalidade, o RSW pôde demonstrar seu poder através de exemplos completos de autômatos adaptativos, sendo estes utilizados na solução de problemas na área de geração automática de compiladores. Alguns destes exemplos mostram como é possível oferecer um tratamento puramente sintático para questões relacionadas com a semântica estática.

O primeiro trabalho explorando a forte relação entre adaptabilidade e aprendizagem computacional enfocou problemas relacionados com o processamento de linguagem natural. Nesse trabalho apresentou-se uma opção bastante interessante para a indução automática de linguagens a partir de exemplos usando autômatos adaptativos (NETO; IWAI, 1998).

Curiosamente, foi na área da música que surgiu a primeira experiência prática de aplicação da tecnologia adaptativa: o LASSUS[2], um gerador automático de composições musicais que lembram os corais barrocos para órgão, executados a quatro vozes (BASSETO; NETO, 1999). Uma característica interessante deste produto é a pouca quantidade de memória ocupada, apenas alguns kilobytes, contendo basicamente um conjunto de regras gerais de composição musical representados e manipulados por redes de Markov adaptativas. A utilização de redes de Markov como mecanismo subjacente também foi uma inovação importante, ao aplicar a tecnologia adaptativa a sistemas envolvendo processos estocásticos. A aleatoriedade faz com que cada execução do LASSUS produza uma composição diferente e imprevisível, de excelente qualidade estética.

As possibilidades de utilização do formalismo adaptativo em aprendizagem computacional e, de maneira mais geral, em inteligência artificial, foram exploradas com maior profundidade em um trabalho que integra conceitos de algoritmos genéticos, redes neurais artificiais e autômatos adaptativos. O resultado desta integração gerou um protótipo de um sistema para busca automática de soluções, o BSMA (Busca de Soluções por Máquina Adaptável) (ROCHA; NETO, 2000c, 2001a, 2000b). Uma demonstração interessante de como o BSMA pode ser utilizado na simulação de redes

[2] Orlande de Lassus foi um dos mais férteis e versáteis compositores do século 16

neurais artificiais foi apresentada em (ROCHA, 2001; ROCHA; NETO, 2001b).

A utilização de autômatos adaptativos em problemas de processamento de linguagens naturais foi novamente explorada em (MENEZES; NETO, 2000). Esse trabalho propõe uma extensão para o modelo de autômatos adaptativos, o autômato adaptativo E (Estendido), em que contadores podem ser associados a cada transição. O produto resultante deste trabalho foi um etiquetador morfológico para a língua portuguesa, cujo desempenho pode ser melhorado através de treinamento a partir de exemplos. A possibilidade de combinar conhecimento prévio com inferência estatística também foi ilustrada nesse trabalho (MENEZES; NETO, 2002).

Autômatos adaptativos, como qualquer outro tipo de autômato, são mecanismos reconhecedores de linguagem. Mas assim como os outros reconhecedores estudados na teoria da computação, os autômatos adaptativos também possuem uma contrapartida entre os mecanismos geradores de linguagem: as gramáticas adaptativas. Tais gramáticas foram introduzidas em (IWAI, 2000; IWAI; NETO, 2000), junto com o teorema de equivalência expressiva entre gramáticas e autômatos adaptativos.

Uma nova contribuição da tecnologia adaptativa para a área de especificação e implementação de sistemas computacionais envolveu a utilização de um autômato adaptativo no desenvolvimento de um ambiente de apoio a programação multilinguagem (FREITAS; NETO, 2000, 2001; NETO; FREITAS, 2001). Esse autômato adaptativo, o coletor de nomes, embora simples, representa uma alternativa bastante eficiente e elegante às estruturas de dados de armazenamento e busca de cadeias. No caso do projeto com multilinguagem de programação, o coletor de nomes foi usado para implementar um dos módulos centrais de um ambiente de programação, o AML (Ambiente MultiLinguagem), no qual as linguagens C++, Prolog, Lisp e Java podiam ser utilizadas concomitantemente.

Na área da robótica, a primeira proposta de utilização do formalismo adaptativo considerou problemas relacionados com o mapeamento de ambientes e planejamento de rotas para navegação autônoma (JUNIOR; NETO; HIRAKAWA, 2000). Nessa proposta, a capacidade intrínseca de auto-modificação de estruturas internas, característica intrínseca dos autômatos adaptativos, foi apresentada como uma forma alternativa às técnicas usuais para a aquisição gradativa de informações em um ambiente geográfico previamente desconhecido. O problema relacionado, de captura de informação através do reconhecimento de padrões em imagens, foi tratada recentemente, em (COSTA; HI-

RAKAWA; NETO, 2002), também utilizando tecnologia adaptativa.

A viabilidade da utilização de autômatos adaptativos na modelagem de problemas relacionados com o processamento de linguagem natural foi consolidada com a demonstração de que dois formalismos tradicionais nesta área, as redes ATN (*Augmented Transitions Network*) e as gramáticas GPSC (*Generalized Phrase Structure Grammar*), podem ser facilmente mapeados em autômatos adaptativos (TANIWAKI; NETO, 2001). A existência de uma especificação da gramática da língua portuguesa em GPSC torna ainda mais importante este mapeamento, pois permite o reaproveitamento de um extenso trabalho de codificação, utilizando agora a tecnologia adaptativa. Um outro trabalho importante, este envolvendo o tratamento adaptativo de indeterminismos e ambigüidades, características bastante comuns em linguagens naturais, é descrito em (NETO; MORAES, 2002).

Os conceitos fundamentais que permeiam as versões formais adaptativas dos autômatos, gramáticas, *statecharts* e redes de Markov foram consolidados e generalizados em (NETO, 2001), com a introdução dos dispositivos adaptativos baseados em regras. Neste trabalho também foi apresentada uma nova instância do formalismo geral dos dispositivos adaptativos: as tabelas de decisão adaptativas. Esta generalização representa um marco importante na história dos dispositivos adaptativos, e abre um novo campo de exploração, que deverá resultar na criação de ferramentas poderosas, combinando e flexibilizando a utilização dos mais diferentes formalismos baseados em regras.

Uma aplicação da tecnologia adaptativa na área de ensino à distância, utilizando uma nova instância de dispositivo adaptativo, o AMBER adaptativo, foi proposta recentemente em (CAMOLESI; NETO, 2002). O modelo AMBER é uma ferramenta de apoio ao projeto de sistemas distribuídos, e sua versão adaptativa aumenta o poder de expressão original, possibilitando a modelagem natural de sistemas com regras que se modificam dinamicamente. O gerenciamento de ambientes de ensino à distância, tratado em (CAMOLESI; NETO, 2002), é um excelente exemplo de um sistema distribuído e altamente dinâmico, cuja especificação pôde ser facilitada com a introdução do AMBER adaptativo.

Um dos mais recentes formalismos a receber uma versão adaptativa foi a árvore de decisão (PISTORI; NETO, 2002). Uma implementação de uma árvore de decisão adaptativa, a AdapTree-E, foi utilizada no desenvolvimento de dois protótipos criados

a partir da integração entre a tecnologia adaptativa, a aprendizagem computacional e a visão computacional. O primeiro protótipo, o vTTT (*TicTacToe by Vision*), é um jogo da velha que pode ser treinado para capturar a direção do olhar do usuário, através de uma câmera filmadora posicionada acima do monitor, e inferir a posição do tabuleiro a ser marcada (PISTORI; NETO; COSTA, 2003a, 2003b). Desta forma, o usuário pode interagir com o sistema sem utilizar as mãos. O segundo protótipo, que obedece princípios bastante similares ao primeiro, é um editor de textos que "entende" um subconjunto dos sinais do alfabeto LIBRAS, para surdos. Neste último caso, o usuário deve posicionar sua mão à frente da câmera filmadora, e executar os sinais correspondes aos caracteres a serem digitados. Um sistema como este, devidamente aprimorado e atrelado a um conversor texto-voz, poderá no futuro ser embutido em um sistema compacto e portável (um telefone celular, por exemplo), a ser utilizado por pessoas que não conheçam o alfabeto LIBRAS, na comunicação com surdos.

A construção de uma ferramenta de apoio ao desenvolvimento de autômatos adaptativos foi retomada em 2003, agora utilizando uma linguagem portável, Java, e uma metodologia de desenvolvimento aberta (com os programas-fonte livres e disponíveis na Internet) e orientada a objetos. O AdapTools (PISTORI; NETO, 2003) já se encontra em uma fase de desenvolvimento adiantada (versão 1.4.3), oferecendo uma variedade de recursos de depuração e animação gráfica, além de diversos exemplos de autômatos implementados. Seguindo uma tendência de simplificação notacional para autômatos adaptativos (NETO; PARIENTE, 2002; PISTORI; NETO, 2002), o AdapTools também adota algumas pequenas modificações, relatadas no manual on-line do software, que se encontra disponível no *site* do grupo de pesquisa em linguagens e tecnologias adaptativas [3].

A figura 2.1 resume os principais eventos ocorridos durante o desenvolvimento da teoria e tecnologia adaptativa descritos acima. A linha de tempo é apenas uma aproximação, baseada na data de publicação de artigos, dissertações e teses referenciando o evento em questão. No lado esquerdo da linha estão as contribuições teóricas; como novos conceitos, definições e formalizações. No outro lado apresenta-se algumas ferramentas, aplicações e protótipos.

[3] http://www.pcs.usp.br/~lta

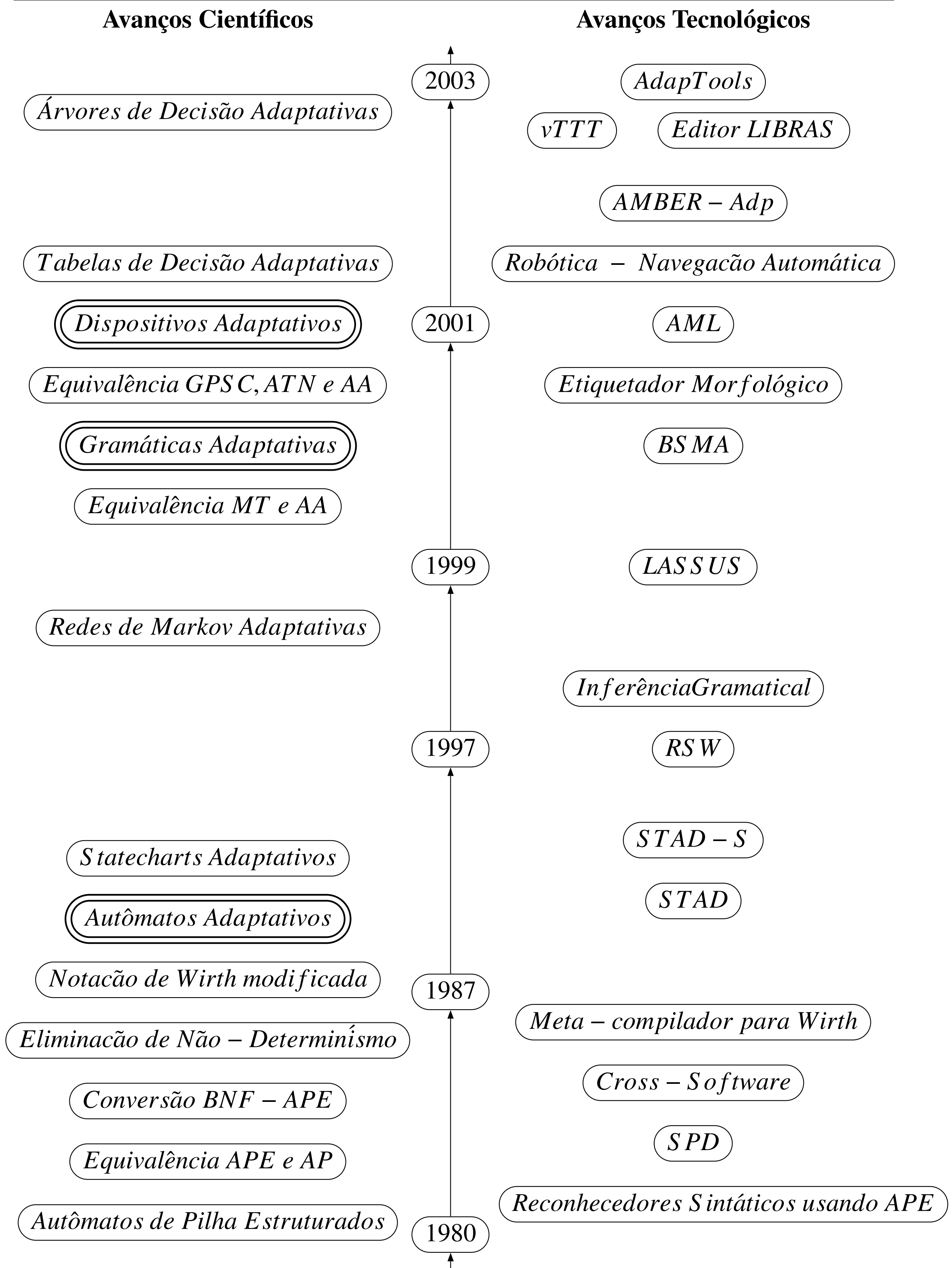


Figura 2.1: Principais acontecimentos nacionais na história dos dispositivos adaptativos

2.1 Formalismos Adaptativos no Contexto Mundial

Os primeiros vestígios dos dispositivos adaptativos, que atingiram sua formalização mais geral em (NETO, 2001), podem ser encontrados nos trabalhos de van Wijngaarden (WIJNGAARDEN, 1974) e suas gramáticas de dois níveis. Embora as gramáticas de dois níveis não possam ser estritamente consideradas dispositivos adaptativos, uma vez que não permitem alteração dinâmica do seu conjunto de regras, elas seguem o mesmo principio de reutilização de formalismos menos poderosos. A idéia revolucionária de van Wijngaarden consistiu, basicamente, em utilizar gramáticas livres de contexto, não para gerar diretamente a linguagem desejada, mas para gerar especificações de outras gramáticas livres de contexto, estas sim representando as construções da linguagem alvo. Gramáticas de dois níveis foram utilizadas para representar de uma maneira elegante a linguagem Algol 68.

A primeira pessoa a apresentar, informalmente, a idéia de estender dinamicamente uma gramática, para tratar problemas relacionados com dependência de contexto, parece ter sido Di Forino (CHRISTIANSEN; SHAW, 1990), em (FORINO, 1963). Entre os dispositivos precursores das gramáticas adaptativas, Christiansen (CHRISTIANSEN; SHAW, 1990) cita também as gramáticas livre de contexto extensíveis (*Extensible Context-Free Grammars*) (WEGBREIT, 1970, 1980), os tradutores baseados em *templates* dinâmicos (*Dynamic Template Translators*) (MASON, 1984, 1987) e um conceito de sintaxe dinâmica baseada em λ-cálculo (HANFORD; JONES, 1973), que aparentemente acabou não sendo formalizado.

Os últimos trinta anos presenciaram o surgimento de diversos tipos de dispositivos adaptativos, todos compartilhando o mesmo princípio, que consiste em adicionar a capacidade de auto-modificação a um formalismo específico, preexistente, visando extrapolar sua limitação expressiva. Entre estes dispositivos estão as gramáticas modificáveis (*Modifiable Grammars*) (BURSHTEYN, 1990b, 1990a), os autômatos finitos auto-modificáveis (*Self-Modifying Finite Automata*) (RUBINSTEIN; SHUTT, 1994, 1995; SHUTT, 1995), as gramáticas dinâmicas (*Dynamic Grammars*) (BOULLIER, 1994), as gramáticas evolutivas (*Evolving Grammars*) (CABASINO; PAOLUCCI; TODESCO, 1992), as gramáticas geradoras (*Generative Grammars*) (CHRISTIANSEN, 1986; CHRISTIANSEN; SHAW, 1990; CHRISTIANSEN, 1993), as gramáticas adaptáveis recursivas (*Recursive Adaptable Grammars*) (SHUTT, 1993) e as recentes Gramáticas Meta-S (§-*Grammars*) (JACKSON; LANGAN, 2001; JACKSON, 2000, 2002). Estudos

aprofundados e comparativos envolvendo grande parte destes formalismos podem ser encontrados em (CHRISTIANSEN; SHAW, 1990; SHUTT, 1993; IWAI; NETO, 2000) [4]

A figura 2.2 mostra, em uma linha de tempo, a época aproximada em que cada um dos principais formalismos adaptativos conhecidos foram apresentados à comunidade científica. Linhas duplas foram utilizadas para destacar as contribuições desenvolvidas no Brasil. Dispositivos geradores e reconhecedores foram posicionados, respectivamente, ao lado esquerdo e direito da linha do tempo. Os dispositivos guiados por regras adaptativos (abreviados para "dispositivos adaptativos") aparecem sobre a linha do tempo porque unificam os conceitos de geração e reconhecimento de linguagens.

2.2 Dispositivos Guiados por Regras Adaptativos

O conceito de *dispositivo guiado por regras* generaliza a formalização de uma série de dispositivos formais, como por exemplo, autômatos finitos, autômatos de pilha e máquinas de Turing, que compartilham a característica fundamental de terem sua operação definida por uma conjunto fixo e finito de regras. Estas regras mapeiam cada possível configuração do dispositivo em uma nova configuração, eventualmente levando em consideração um determinado *estímulo de entrada* e gerando algum *símbolo de saída*. Um dispositivo guiado por regras inicia seu funcionamento em uma determinada configuração, e segue aplicando sucessivamente uma regra do seu conjunto de regras, alternando entre as possíveis configurações, até que não existam mais estímulos de entrada ou até que se atinja uma configuração à qual nenhuma regra possa ser aplicada. Neste ponto determina-se, com base na configuração atingida, se o dispositivo aceita ou rejeita a seqüência completa de estímulos de entrada. Assume-se que o conjunto de todas as configurações possa então ser particionado em configurações de *aceitação* e *rejeição* (NETO, 2001).

Em um *dispositivo guiado por regras adaptativo*, ou simplesmente *dispositivo adaptativo*, o conjunto de regras passa a poder variar durante a leitura dos estímulos de entrada. Esta variação, no entanto, é completamente determinada por um outro nível de regras, denominadas *ações adaptativas*, que agem sobre o conjunto de regras original, modificando-o através da remoção e inserção de novas regras. Temos assim um dispositivo com duas camadas, a primeira, denominada *camada subjacente*, é representada

[4] O trabalho de Shutt possui uma versão atualizada disponível na Internet, no endereço http://www.cs.wpi.edu/~jshutt/thesis/top.html

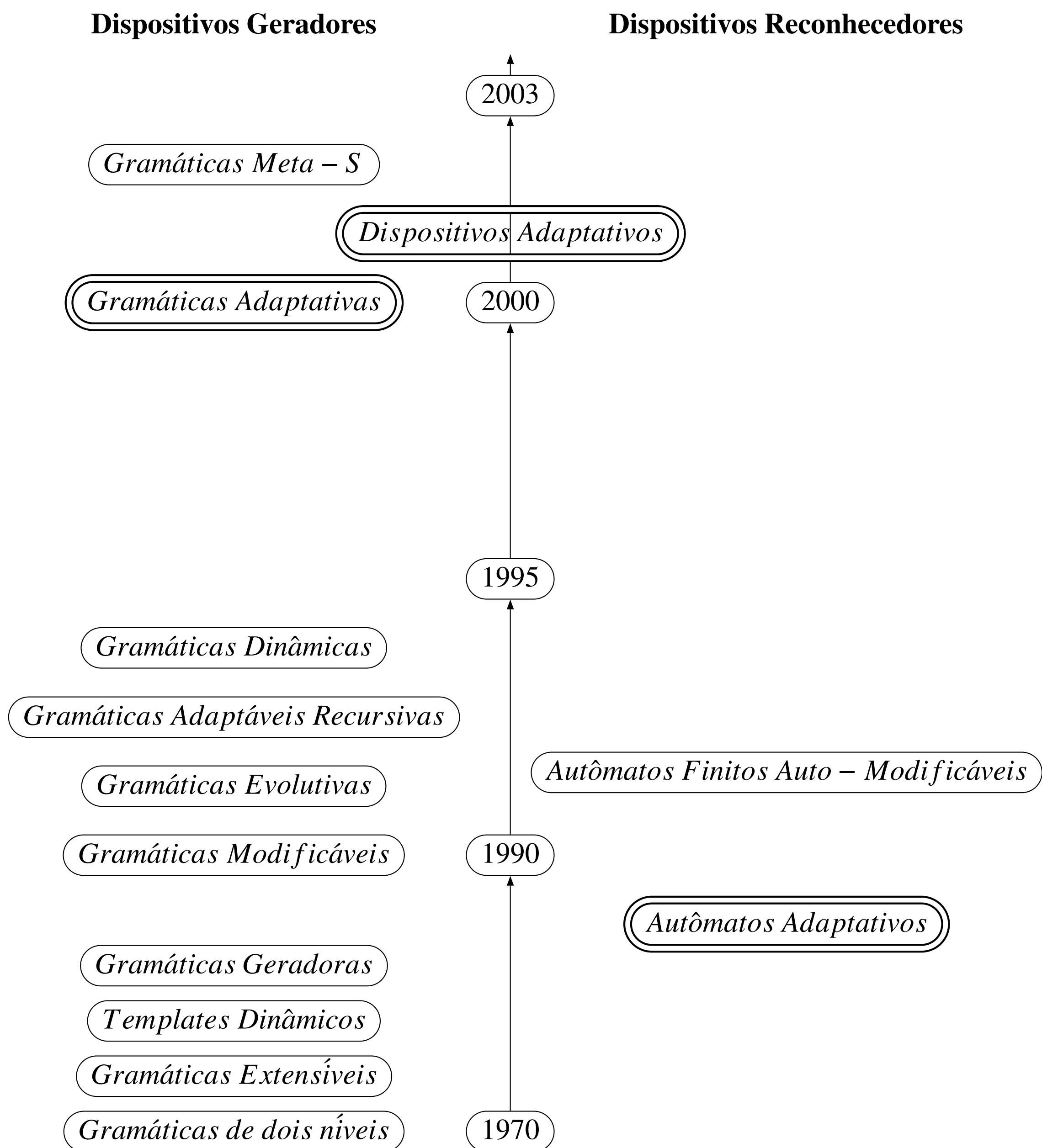


Figura 2.2: Linha de Tempo para Formalismos Adaptativos

por um dispositivo guiado por regras (não-adaptativo), e a segunda, denominada *camada adaptativa*, é definida por um conjunto de ações adaptativas. Com isto, define-se um mecanismo universal, a camada adaptativa, capaz de transformar um dispositivo qualquer, não adaptativo, mas guiado por regras, em um dispositivo capaz de alterar sua estrutura interna (conjunto de regras) durante sua operação.

Dispositivos adaptativos podem ser formalizados como uma dupla $DA = (CS_0, CA)$, em que $CS_0 = (\mathscr{C}_0, \Sigma, \Phi, c_0, c_A, R_0)$ representa um dispositivo guiado por regras não-adaptativo, a camada subjacente, em sua situação original; e $CA = (\mathscr{R}, \mathscr{A})$ representa a camada adaptativa. Os elementos da camada subjacente, CS_0, são descritos da seguinte forma:

$\boldsymbol{\mathscr{C}_0 \subseteq \mathscr{C}}$ é o conjunto das possíveis configurações da camada subjacente em sua situação inicial. Estas configurações são extraídas do conjunto $\mathscr{C}$, que contém todas as possíveis configurações para o DA.

$\boldsymbol{\Sigma}$ é um conjunto fixo e finito contendo todos os possíveis eventos válidos como estímulo de entrada para DA. Este conjunto inclui o símbolo ϵ, utilizado aqui (e no resto desta seção) para representar explicitamente um valor nulo.

$\boldsymbol{\Phi}$ é um conjunto fixo e finito de possíveis símbolos de saída, incluindo o símbolo ϵ acima citado.

$\boldsymbol{c_0 \in \mathscr{C}_0}$ representa a configuração inicial do dispositivo.

$\boldsymbol{c_A \subseteq \mathscr{C}}$ é o conjunto de configurações de aceitação (as configurações de rejeição, c_R, são definidas pelo complemento de c_A em relação à $\mathscr{C}$).

$\boldsymbol{R_0 \subseteq R \subseteq \mathscr{C} \times \Sigma \times \mathscr{C} \times \Phi}$ é o conjunto de regras da camada subjacente em sua situação inicial (antes de qualquer aplicação de ações adaptativas). O conjunto R, de maneira análoga à $\mathscr{C}$, contém todas as possíveis regras de um DA (ou seja, contém além das regras iniciais, aquelas que poderão vir a ser inseridas durante a execução de uma ação adaptativa). Cada regra $r = (c_i, s, c_j, z) \in R$ deve ser interpretada da seguinte forma: dado um estímulo de entrada $s \in \Sigma$ e estando na configuração $c_i \in \mathscr{C}$, passa para a configuração $c_j \in \mathscr{C}$, consumindo s, e produzindo $z \in \Phi$.

Os elementos da camada adaptativa, CA, são definidos assim:

$\mathscr{R}$ é um conjunto de ações adaptativas, que contém também o valor nulo ϵ (representando ações que não causam alterações na camada subjacente).

$\mathscr{A}: \boldsymbol{R} \rightarrow \mathscr{R}^2$ é uma função que mapeia cada possível regra da camada subjacente [5] em um par ordenado de ações adaptativas. As duas ações adaptativas deste par ordenado devem ser acionadas, respectivamente, antes e depois da execução da regra à qual elas estão associadas. Por isto, elas são também denominadas, respectivamente, *ação anterior* e *ação posterior*.

A execução das ações adaptativas induzem uma seqüência, $CS_0, CS_1, CS_2, ..., CS_n$, de transformações da camada subjacente, na qual cada elemento elemento $CS_i, 0 \leq i \leq n$ é definido de maneira análoga à CS_0. Os elementos $\mathscr{C}_i$ e R_i, de CS_i, correspondem aos conjuntos de configurações e regras produzidos pela aplicação da uma ação adaptativa a_i $(i > 0)$, sobre CS_{i-1}. A operação de um dispositivo adaptativo pode ser resumida através do algoritmo 2.1.

2.2.1 Autômatos Adaptativos

Um autômato adaptativo é um dispositivo guiado por regras adaptativo em que a camada subjacente consiste de um autômato de pilha estruturado (ver anexo A) e as ações adaptativas da camada adaptativa são implementadas através de *funções adaptativas*. Nesta seção faremos uma revisão da definição original de uma função adaptativa, como apresentada em (NETO, 1993).

As funções adaptativas determinam exatamente quais modificações devem ser realizadas na camada subjacente do dispositivo, quando uma ação adaptativa é acionada. De certa forma, podemos entender uma ação adaptativa como uma chamada de função, a função adaptativa, que pode, inclusive, ser paramétrica. O núcleo de uma função adaptativa consiste de uma lista de *ações adaptativas elementares*. São três os tipos de ações adaptativas elementares: as *ações elementares de consulta*, que possibilitam a busca de padrões na estrutura definida pelas regras da camada subjacente; e as *ações elementares de inserção* e de *remoção*, que determinam, respectivamente,

[5] Ao definir que o domínio desta função contêm também regras que poderão vir a ser introduzidas durante a execução do dispositivo, conseguimos isolar completamente a camada subjacente da camada adaptativa. No entanto, a especificação de ações adaptativas que inserem regras que devem, por sua vez, serem associadas a outras ações adaptativas, depende agora de uma especificação apropriada da função $\mathscr{A}$

Algoritmo 2.1 Operação de um dispositivo adaptativo

1: Posicione o dispositivo em sua configuração inicial, c_0.
2: Prepare o dispositivo para receber o primeiro estímulo de entrada.
3: **enquanto** Existerem estímulos de entrada **faça**
4: Sejam $s \in \Sigma$ e $c_T \in \mathscr{C}_T$ o estímulo de entrada e a configuração correntes, respectivamente.
5: Seja $\delta_T \subseteq R_T$ o conjunto de regras que podem ser aplicadas na situação corrente.
6: Crie um conjunto Δ_T contendo cada uma das regras de δ_T acrescidas das ações anteriores e posteriores, a_a e a_p, definidas por $\mathscr{A}$. Cada elemento de Δ_T terá a forma (c_T, s, c, z, a_a, a_p), com $c_T, c \in \mathscr{C}$, $s \in \Sigma$ e $z \in \Phi$.
7: **se** $|\Delta_T| = 0$ **então** {Nenhuma regra compatível}
8: Terminar operação rejeitando a entrada.
9: **senão se** $|\Delta_T| = 1$ **então** {Exatamente uma regra compatível}
10: Seja $r = (c_T, s, c, z, a_a, a_p)$ o único elemento de Δ_T
11: **senão se** $|\Delta_T| > 1$ **então** {Mais de uma regra compatível}
12: Prossiga a execução do algoritmo "paralelamente" para cada elemento de Δ_T, fazend $r = (c_T, s, c, z, a_a, a_p)$ corresponder a um diferente elemento de Δ_T em cada uma das diferentes instâncias do algoritmo.
13: **fim se**
14: Seja CS_T a situação atual da camada subjacente.
15: **se** $a_a \neq \epsilon$ **então**
16: Aplique a ação adaptativa a_a gerando assim CS_{T+1}.
17: **se** a regra sendo aplicada não existe mais em CS_{T+1} **então**
18: Retorne ao passo 4
19: **fim se**
20: **fim se**
21: Aplique a regra r (considerando apenas os 4 primeiro elementos, referentes à camada subjacente), alterando assim a configuração da camada subjacente, de c_T para $c_{T+1} = c$, consumindo s e produzindo z.
22: **se** $a_p \neq \epsilon$ **então**
23: Aplique a ação adaptativa a_p gerando assim CS_{T+1} (ou CS_{T+2}, caso a ação anterior também tenha sido aplicada).
24: **fim se**
25: **fim enquanto**
26: **se** Configuração corrente é de aceitação **então**
27: Aceite a seqüência de estímulos
28: **senão**
29: Rejeite a seqüência de estímulos
30: **fim se**

as regras que deverão ser inseridas ou removidas do conjunto de regras corrente da camada subjacente. Formalmente, define-se uma função adaptativa como uma 9-upla $FA = (F, P, V, G, C, E, I, A, B)$ na qual:

F é o nome da função adaptativa.

P é a lista $(r_0, r_1, ..., r_m)$ de parâmetros formais.

V é a lista $(v_1, v_2, ..., v_n)$ de identificadores de variáveis.

G é a lista $(g_1^*, g_2^*, ..., g_p^*)$ de identificadores de geradores.

C é a lista de ações de consulta.

R é a lista de ações de remoção.

I é a lista de ações de inserção

A é uma *ação adaptativa inicial*, opcional, que deve ser executada antes de F.

B é uma *ação adaptativa final*, opcional, que deve ser executada depois de F.

Todas as ações adaptativas são da forma (F', P'), na qual:

F′ é o nome de uma função adaptativa.

P′ é a lista $(p_0, p_1, ..., p_k)$ de argumentos a serem passados para F'.

A execução de uma função adaptativa inicia com um mecanismo de passagem de parâmetros por valor automaticamente atribuindo, a cada parâmetro formal, r_i, o valor atualmente atribuído ao argumento p_i, em P'. Depois disto ocorre a execução da ação adaptativa inicial (opcionalmente), que pode repassar, se necessário, os valores dos parâmetros recebidos por F, para uma outra função adaptativa. A ação adaptativa inicial, no entanto, não tem acesso a variáveis e geradores, pois seus valores ainda não estão definidos neste ponto da execução da função adaptativa.

Ações elementares possuem basicamente a forma de uma regra da camada subjacente, possivelmente contendo nomes de variáveis e de geradores no lugar de alguns dos elementos da regra. Na execução das ações elementares de consulta, uma mecanismo de busca por padrões é responsável por atribuir valores à estas variáveis, tendo

como base o conjunto de regras do mecanismo subjacente. Uma vez atribuidos, os valores de cada variável não podem mais ser alterados durante a execução da função adaptativa. Quando o mecanismo de busca por padrões não encontra qualquer regra na camada subjacente que satisfaça o formato determinado pela ação elementar de consulta, as variáveis permanecem indefinidas.

A execução de ações elementares de remoção segue, inicialmente, o mesmo princípio da execução das ações elementares de consulta, com uma busca de padrões sendo utilizada para determinar valores para variáveis (que podem também estar presentes nas ações elementares de remoção). Após a busca, caso todos os valores de variáveis tenham sido definidos, a regra correspondente deve ser eliminada da camada subjacente (caso contrário - existem variáveis indefinidas -, a ação elementar é simplesmente ignorada).

Para a execução das ações elementares de inserção, que ocorre sempre depois da execução das ações de consulta e remoção, todas as variáveis devem ter sido previamente instanciadas (ou marcadas como indefinidas). Para os casos em que todas as variáveis contidas na ação elementar de inserção estão definidas, procede-se a inserção das regras correspondentes na camada subjacente. Ações elementares de inserção podem também conter nomes de geradores, ao invés de variáveis. Neste caso, antes da inserção da nova regra na camada subjacente, todos os geradores são substituídos por novos símbolos, diferentes de qualquer símbolo utilizado na camada subjacente.

Por fim, executa-se a ação adaptativa final, que pode agora fazer referência à variáveis e geradores.

Em (NETO, 1993, 1994) é proposta uma notação para funções adaptativas que possui basicamente o formato abaixo, com “padrão” sendo uma estrutura dependente do formato das regras da camada subjacente:

$F(r_1, ..., r_m) = \{$

$v_1, v_2, ..., v_n, \quad g_1^*, g_2^*, ..., g_p^* :$

$A(p_1, ..., p_k)$

$?[padrão]$

$?[padrão]$

..

$-[padrão]$

$-[padrão]$

..

$+[padrão]$

$+[padrão]$

..

$B(p_1, ..., p_{k'})$

$\}$

Os prefixos ?, $-$ e + denotam ações elementares de consulta, remoção e inserção, respectivamente. Um exemplo de um autômato adaptativo escrito através desta notação será apresentado no capítulo 8 (seção 8.1.1), juntamente com a versão modificada que pode ser executada pela ferramenta AdapTools (descrita também no capítulo 8).

3 APRENDIZAGEM DE MÁQUINA

Nos paradigmas tradicionais de programação de computadores, como os imperativos, funcionais ou orientados por objetos, cabe inteiramente aos desenvolvedores obter uma representação computacional implementável da solução do problema a ser resolvido. A aprendizagem computacional (MITCHELL, 1997), ou aprendizagem de máquina, oferece aos engenheiros e cientistas de computação um paradigma, para desenvolvimento de software, bastante diferente dos tradicionais. Em linhas gerais, neste paradigma não cabe mais ao desenvolvedor a tarefa de encontrar diretamente a solução para um problema específico, mas apenas a de criar um ambiente computacional que permita que esta solução seja automaticamente (ou semi-automaticamente) induzida a partir de exemplos de como instâncias particulares do problema são resolvidas.

Outro ponto interessante do paradigma da aprendizagem computacional é que o mesmo ambiente de aprendizagem pode ser utilizado, sem alterações, na solução de problemas diferentes do originalmente proposto. Ainda não se conhecem estratégias de aprendizagem computacional que consigam sequer imitar a generalidade do processo de aprendizagem que ocorre nos seres humanos. No entanto, soluções para problemas reais, desenvolvidas a partir de técnicas de aprendizagem de máquina, já começam a ser aplicadas nas mais diversas áreas, como diagnóstico médico, interpretação de linguagem natural e reconhecimento óptico de caracteres (OCR) (MICHALSKI; BRATKO; KUBAT, 1998).

Uma das maneiras de classificar as diferentes técnicas de aprendizagem de máquina refere-se à classe do modelo utilizado para representar a solução induzida a partir dos exemplos. Entre as principais classes de modelos utilizados estão as árvores de decisão (KOTHARI; DONG, 2001), as redes neurais artificiais (HAYKIN, 1999), as sentenças em lógica de predicados (MUGGLETON; RAEDT, 1994), os conjuntos de re-

gras “se-então” (CLARK; NIBLETT, 1989), os autômatos (PAREKH; HONAVAR, 2001), as redes bayesianas (HECKERMAN; GEIGER; CHICKERING, 1995) e até mesmo a simples “memorização” de parte ou de todos os exemplos utilizados nessa aprendizagem (*instance-based learning*) (AHA; KIBLER; ALBERT, 1991). Informalmente, estes modelos podem ser ordenados quanto à facilidade por parte de um usuário comum (que não domine técnicas avançadas de estatística, teoria dos grafos e matemática em geral) de compreenderem a solução induzida (KANDOLA J.S. E GUNN, 2000). Por exemplo, redes neurais artificiais são vistas, geralmente, como “caixas-pretas”, pois não oferecem uma interpretação facilmente inteligível, para um usuário comum, da solução induzida (KANDOLA J.S. E GUNN, 2000). Já as regras “se-então”, em geral, podem ser compreendidas com uma certa facilidade por qualquer pessoa, sem a necessidade do domínio de conceitos complexos.

A facilidade com que se pode compreender um modelo automaticamente produzido por um algoritmo de aprendizagem é denominada *transparência* desse modelo (KANDOLA J.S. E GUNN, 2000), e pode ser fundamental para o sucesso prático de um sistema computacional. Especialistas da área médica, por exemplo, dificilmente aceitariam uma solução, proposta por um sistema de aprendizagem automática, que não pudesse ser compreendida e justificada através de uma linguagem que seja o mais próxima possível da linguagem natural. Além da transparência, existem diversas métricas a partir das quais a qualidade de uma estratégia de aprendizagem pode ser avaliada. Uma destas métricas, bastante utilizada na prática, é a taxa de acerto, que busca quantificar a capacidade de generalização de um mecanismo de aprendizagem, aplicando-o sobre um conjunto de exemplos diferente daquele usado no treinamento, e medindo o total de exemplos corretamente classificados. O que muitas vezes ocorre é a existência, para um mesmo problema, de uma solução de baixa transparência, mas com alta taxa de acerto, e de outra, transparente, mas com menor taxa de acerto.

Existem pelo menos duas estratégias básicas para tratar o problema da falta de transparência. A primeira, e mais antiga, consiste em criar mecanismos de conversão entre diferentes modelos, como por exemplo, extrair regras “se-então” a partir de árvores de decisão. A vantagem desta estratégia é que o algoritmo original de aprendizagem é completamente preservado. No entanto, existe sempre um custo computacional adicional para se efetuar esta conversão. A segunda estratégia propõe novos algoritmos de aprendizagem que tentam reunir, em um único formalismo, as características mais desejáveis de dispositivos já existentes, os quais podem ou não sofrer alterações.

Estas soluções são geralmente denominadas *híbridas*, e podem gerar, por exemplo, árvores de decisão cujos nós são redes neurais (FRANK et al., 1998; KOTHARI; DONG, 2001), ou ainda, redes neurais artificiais especialmente moldadas para se comportarem como autômatos de estados finitos (GILES et al., 1995; OMLIN; GILES, 1996). Estratégias híbridas podem representar uma ponte entre dois universos usualmente explorados independentemente na área da inteligência artificial (IA): a IA simbólica, baseada principalmente em modelos derivados da matemática discreta (lógica, linguagens formais, grafos, etc) e geralmente mais transparentes; e a IA sub-simbólica, inspirada em modelos biológicos do cérebro humano (redes neurais artificiais) e na teoria da evolução da espécies (algoritmos genéticos), geralmente menos transparentes e com utilização mais ampla de conceitos provenientes das áreas de estatística e cálculo.

3.1 Aprendizagem de Máquina e Árvores de Decisão

Árvores de decisão (ADs) são mecanismos para a representação de funções discretas sobre múltiplas variáveis (contínuas ou discretas) com características hierárquicas que facilitam a inspeção e a utilização por seres humanos. Vértices internos de uma árvore de decisão representam testes a serem efetuados sobre alguma variável X, e de cada vértice parte uma aresta para cada possível valor assumido por X. O conjunto imagem da função é representado pelas folhas da árvore. Por exemplo, qualquer uma das árvores de decisão apresentadas na figura 3.1 podem ser utilizadas para representar a função, $f : N \times L \to C$, mostrada na tabela 3.1, em que $N = \{0, 1, 2\}$, $L = \{a, b\}$ e $C = \{sim, não\}$.

N	L	f(N,L)
0	*a*	*sim*
0	*b*	*sim*
1	*a*	*não*
1	*b*	*não*
2	*a*	*sim*
2	*b*	*não*

Tabela 3.1: Função f para o exemplo de árvore de decisão

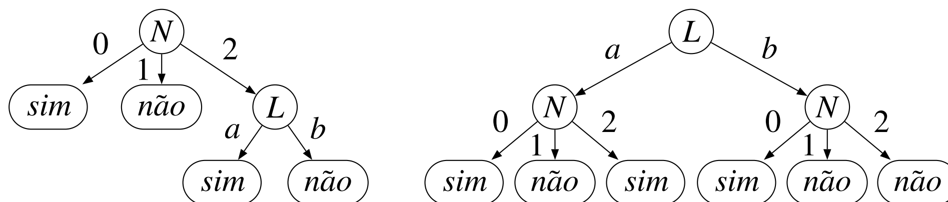


Figura 3.1: Duas árvores de decisão representando a mesma função f

No caso de variáveis contínuas, os testes são escolhidos de tal forma que parti-

cionem o domínio em intervalos (e.g. $X \leq 2.23$ e $X \geq 5.6$). Existem também algumas extensões das árvores de decisão que podem ser utilizadas na representação de funções contínuas (no contradomínio), entre as quais podemos citar as árvores de regressão (QUINLAN, 1992) e as árvores de modelos (*Model Trees*) (FRANK et al., 1998). Estas árvores, no entanto, não foram exploradas no presente trabalho.

Um problema importante na área da aprendizagem de máquina consiste em encontrar algoritmos capazes de construir uma AD que represente uma função desconhecida f, dado um subconjunto T, em geral impróprio, de f. A função f é também denominada, no jargão da área, *conceito-alvo* e o resultado produzido pelo algoritmo de indução é dito *representação do conceito*. Além disto, cada elemento de T é denominado *vetor de atributos*, ou *exemplo do conceito*, enquanto o subconjunto T constitui o *conjunto de treinamento* do algoritmo. Em geral, o conjunto de treinamento é fornecido como entrada para o algoritmo de indução, que produz como saída uma representação do conceito (e.g. árvore de decisão, rede neural). Esta representação do conceito pode ser posteriormente utilizada para calcular a função f sobre elementos que não estavam presentes no conjunto de treinamento, ou, em outras palavras, para *classificar* novos exemplos do conceito aprendido.

A maioria dos algoritmos de indução de AD segue uma mesma estratégia geral, que consiste em construir a árvore da raiz para as folhas, particionando o conjunto de treinamento de acordo com os testes escolhidos para cada vértice. Cada teste funciona assim como um filtro no conjunto de treinamento, de forma que a decisão sobre quais testes devem ser associados aos vértices próximos às folhas da árvore é geralmente baseada em um conjunto bem menor de exemplos que aqueles próximos à raiz. Por isso, costuma-se dizer que a construção da árvore de decisão segue um princípio de otimização localizada (KOTHARI; DONG, 2001). A estratégia utilizada para comparar atributos (e.g. *information gain* (QUINLAN, 1993), *chi-square statistic* (MINGERS, 1987)) é um dos fatores que distinguem os diversos algoritmos de indução. Em geral, essa estratégia é construída com o objetivo de tornar mais provável a obtenção da menor árvore de decisão (em número de vértices) capaz de representar corretamente os exemplos recebidos como entrada. A preferência por árvores de decisão mais simples baseia-se no princípio da *Occam Razor* (FORSTER, 2001), o que sugere que, dentre explicações igualmente satisfatórias, são as mais simples que devem ser escolhidas [1]. No entanto, é possível demonstrar que a preferência por árvores de de-

[1] Um estudo sistemático e moderno relacionado a este princípio pode ser encontrado em (FORSTER,

cisão mais simples nem sempre oferece bons resultados, sendo apenas mais um tipo de tendência (*bias*), como por exemplo, a opção por árvores com muitos nós (SCHAFFER, 1991, 1994; ANDERSSON; DAVIDSSON; LIND'EN, 1998).

Um outro fator distintivo dos diversos algoritmos de indução de AD é a maneira como o ponto em que o algoritmo deve interromper a expansão da árvore (*prunning*) é determinado. Árvores muito ramificadas podem apresentar um problema muito conhecido, denominado *overfitting*, que ocorre quando um modelo é tão aderente ao conjunto de treinamento que, por ser demasiadamente específico, resulta inútil para os exemplos externos a este conjunto; em outras palavras, o modelo foi incapaz de generalizar a solução encontrada. Uma das maneiras mais comuns para se verificar a capacidade de generalização do modelo é utilizar, além do conjunto de treinamento, um conjunto de teste que seja independente do conjunto de treinamento, na avaliação da taxa de acerto do modelo inferido. Uma das estratégias mais conhecidas de *prunning* consiste justamente em avaliar a capacidade de generalização do modelo a cada novo nó criado, parando a recursão assim que a capacidade começa a se deteriorar (MITCHELL, 1997). Uma descrição comparativa entre diversas técnicas de *prunning* e seleção de atributos pode ser encontrada em (KOTHARI; DONG, 2001).

Um algoritmo de aprendizagem computacional é dito incremental quando os exemplos de treinamento podem ser fornecidos um a um, sem que o modelo induzido tenha que ser reconstruído a cada novo exemplo (o modelo é construído incrementalmente). A vantagem de um algoritmo incremental é obvia em ambientes em que um conjunto de exemplos significativo não pode, por natureza ou por dificuldades técnicas, ser obtido antecipadamente; e além disto, o sistema não pode ser interrompido para retreinamento (por exemplo, um robô em sua primeira missão espacial em uma planeta desconhecido). Tradicionalmente, os algoritmos de indução de AD não são incrementais (e.g. ID3 (QUINLAN, 1996) e C4.5 (QUINLAN, 1993, 1996)), no entanto, existem algumas propostas, como os algoritmos ID5 e ITI (UTGOFF; CLOUSE, 1997), que buscam estender a capacidade de algoritmos tradicionais, permitindo que a árvore possa ser incrementalmente re-estruturada após a leitura de cada exemplo.

2001).

4 CÁLCULO DE PREDICADOS E UNIFICAÇÃO

Embora o foco de nossa proposta para a formalização de funções adaptativas, que será apresentada no capítulo 6, seja a unificação e a satisfação seqüencial de restrições, optamos por apresentar neste capítulo uma revisão sobre o cálculo de predicados. Acreditamos que esta revisão facilitará a introdução da satisfação seqüencial de restrições, na seção 4.2, que é apenas uma especialização do cálculo de predicados.

O cálculo, ou lógica, de predicados é um formalismo utilizado para expressar, de maneira não-ambígua, o conhecimento sobre objetos e relações entre objetos que sejam válidas em uma data teoria (GENESERETH; NILSSON, 1988). O termo objeto, utilizado aqui, deve ser entendido em seu sentido mais amplo, podendo incluir, por exemplo, números, cores, sensações, personagens imaginários, programas de computador, palavras reservadas de uma linguagem de programação, etc. Relações entre objetos podem ser compreendidas no sentido utilizado na teoria dos conjuntos. Objetos podem também ser referenciados indiretamente, através do uso de funções. A contrapartida em cálculo de predicados para objetos, funções e relações são as *constantes objeto*, as *constantes funcionais* e as *constantes relacionais* (também chamadas *predicados*). A tabela 4.1 mostra alguns exemplos de sentenças envolvendo objetos, funções e relações, no domínio da aritmética, acompanhados de uma possível representação em cálculo de predicados. Exemplos envolvendo um domínio menos formal são mostrados na tabela 4.2.

A utilização de uma mesma notação para expressões envolvendo predicados e expressões envolvendo constantes funcionais, ao lado de uma analogia imperfeita com conceitos similares da teoria dos conjuntos, costuma confundir o iniciante ao cálculo de predicados. Ao contrário do que ocorre com funções e relações na teoria dos conjuntos, uma constante funcional não deve ser vista como um conceito que especializa o con-

Conceitualização	Cálculo de Predicados
Objetos	
2	2
10	10
3.14159...	Pi
Objetos e Funções	
2^3	Potência(2,3)
$2^3 + 4 \times 5$	Soma(Potencia(2,3),Multiplica(4,5))
Objetos e Relações	
$5 \geq 3$	MaiorIgual(5,3)
3.14159... < 4	MenorQue(Pi,4)
Objetos, Funções e Relações	
$10 > \sqrt{2^2}$	MaiorQue(10,RaizQuadrada(Potencia(2,2)))
$1 < 1 + 2 + 3 + 4$	MenorQue(1,Soma(1,Soma(2,Soma(3,4))))

Tabela 4.1: Exemplos aritméticos envolvendo objetos, funções e relações

ceito de predicado. Expressões funcionais servem para referenciar objetos, enquanto expressões predicativas, ou *átomos* (sentenças atômicas), podem ser entendidas como afirmações sobre as quais podemos associar os valores falso ou verdadeiro. Assim, a expressão funcional $Capital(Brasil)$, refere-se indiretamente a Brasília, enquanto o átomo $Capital(Brasil, BuenosAires)$ é simplesmente uma afirmação, que pode ser falsa ou verdadeira, dependendo da interpretação (a afirmação poderia ser verdadeira, por exemplo, em um livro de ficção). É importante notar que expressões funcionais podem ser utilizadas como argumentos de expressões predicativas, no entanto, o inverso não é verdade.

Átomos podem ainda ser combinados para formar sentenças mais complexas, utilizando os operadores lógicos de negação ($\neg$), conjunção ($\wedge$), disjunção ($\vee$), implicação ($\Rightarrow$), implicação reversa ($\Leftarrow$) e implicação bidirecional ($\Leftrightarrow$). Informalmente, podemos dizer que o significado destes operadores é equivalente ao utilizado em circuitos digitais e em lógica proposicional (uma lógica menos expressiva, que não permite nem a utilização de relações, nem dos quantificadores, que serão comentados adiante). A tabela 4.3 mostra exemplos de sentenças envolvendo operadores lógicos, também denominadas sentenças lógicas.

Dois operadores, os quantificadores universal ($\forall$) e existencial ($\exists$), juntamente com a possibilidade de utilização de variáveis no lugar de constantes objeto e funcionais (mas não no lugar de constantes relacionais), completam o repertório conceitual do

Conceitualização	Cálculo de Predicados
Objetos	
guitarra	Guitarra
Alcides	Alcides
Hemerson	Hemerson
branca	Branca
Objetos e Funções	
Orientador do Hemerson	Orientador(Hemerson)
Pai do pai do Hemerson	Pai(Pai(Hemerson)))
Capital do Brasil	Capital(Brasil)
Idade da Capital do Brasil	Idade(Capital(Brasil))
Filho do Advogado da Julia	Filho(Advogado(Julia))
Objetos e Relações	
Amazonas é maior que Sergipe	Maior(Amazonas,Sergipe)
Guitarra é um instrumento musical	InstrumentoMusical(Guitarra)
José emprestou seu livro de cálculo para Maria	Emprestou(Jose,LivroCalculoDoJose,Maria)
Os estados p e q estão ligados por uma transição que lê b e executa a função adaptativa f	Transicao(Ep,Eq,Sb,Af)
Objetos, Funções e Relações	
O pai de José é mais alto que o de Maria	MaisAlto(Pai(Jose),Pai(Maria))
José gosta da irmã de Maria	Gosta(Jose,Irma(Maria))

Tabela 4.2: Exemplos gerais envolvendo objetos, funções e relações

Conceitualização	Cálculo de Predicados
O pai de Jonas, além de professor, era músico.	Professor(Pai(Jonas)) ∧ Músico(Pai(Jonas))
Se o clima de Campo Grande estiver quente Eva vai nadar ou mergulhar	Quente(Clima(Cgr)) ⇒ (Nadar(Eva) ∨ Mergulhar(Eva))
Eu vou somente se você não for	Vai(Eu) ⇔ ¬ Vai(Você)
Rapadura é doce mas não é mole não	Doce(Rapadura) ∧ (¬ Mole(Rapadura))

Tabela 4.3: Exemplos de sentenças lógicas

Conceitualização	Cálculo de Predicados
Todos são culpados	$\forall x$ Culpado(x)
Existe um culpado	$\exists x$ Culpado(x)
Tudo o que sobe desce	$\forall x$ (Sobe(x) $\Rightarrow$ Desce(x))
Nem tudo que reluz é ouro	$\exists x$ Reluz(x) $\wedge$($\neg$ Ouro(x))
Nem só de pão vive o homem	$\forall x$ Homem(x) $\Rightarrow$ ($\exists y$ Necessita(x,y) $\wedge$ $\neg$ Pão(y))
Todas as amigas de Maria moram em Santos	$\forall x$ Amiga(Maria,x) $\Rightarrow$ Mora(x,Santos)

Tabela 4.4: Exemplos de sentenças quantificadas

cálculo de predicados. Os quantificadores tornam possível a expressão concisa de afirmações a respeito de conjuntos de objetos, o que não é possível em lógicas mais simples, como a proposicional. O quantificador universal captura informações que, em linguagem natural, são introduzidas através dos pronomes indefinidos *todo* e *qualquer*, enquanto o quantificador existencial, pretende capturar o sentido do verbo *existir* e similares. A tabela 4.4 mostra exemplos de utilização de quantificadores, enquanto a gramática, na notação de Wirth, apresentada na figura 4.1, resume a sintaxe de uma fórmula bem formada (*well-formed formulas - wff*) em cálculo de predicados.

Além de um mecanismo para representação de conhecimento, uma lógica oferece também mecanismos formais que permitem que conclusões possam ser derivadas de premissas: as estratégias de inferência. Uma estratégia de inferência é um procedimento bem definido determinando as maneiras como padrões de combinação de sentenças (*regras de inferência*) podem ser aplicados na geração de novas sentenças. Um exemplo clássico de regra de inferência é o *Modus Ponens*, que pode ser utilizado para combinar uma implicação $\Phi \Rightarrow \Psi$, com uma sentença Φ, para formar a nova sentença Ψ. Uma leitura informal desta regra de inferência poderia ser: se Φ implica em Ψ, e Φ ocorre, então Ψ também ocorre.

Dizemos informalmente que uma sentença Ψ é uma *implicação lógica* de um conjunto de sentenças Δ se e somente se não for possível que Ψ seja falso quando Δ for verdadeiro (se alguém concorda com as afirmações em Δ deve, de acordo com a implicação lógica, concordar com Ψ). Existem duas características muito desejáveis para um estratégia de inferência: a corretude (*soundness*) e a completude (*completeness*) [1]. Estratégias corretas garantem que todas as conclusões deriváveis de uma base

[1]O termo completude deve ser entendido neste contexto conforme a definição de Genereseth (GENESERETH; NILSSON, 1988), e não como empregado por Gödel, em seu famoso teorema da incomple-

```
Formula             = FormAtomica | FormLogica | FormQuantificada.

FormAtomica         = Predicado ”(” Termo { ”,” Termo } ”)”.
Predicado           = Maiuscula { Letra }.
Termo               = Objeto | Funcional | Variavel.
Objeto              = ( Maiuscula { Alfanum }) | Numero .
Funcional           = Funcao ”(” Termo { ”,” Termo } ”)”.
Funcao              = Maiuscula { Letra }.
Variavel            = Minuscula.

FormLogica          = ( Formula OperadorBinario Formula ) | ( ”¬” Formula ).
OperadorBinario     = ( ”∧” | ”∨” | ”→” | ”←” | ”⇔”).

FormQuantificada    = Quantificador Variavel Formula.
Quantificador       = ( ”∀” | ”∃”).
Maiuscula           = A | B | ... | Z.
Minuscula           = a | b | ... | z.
Alfanum             = Letra | Digito.
Letra               = Maiuscula | Minuscula.
Numero              = Digito { Digito }.
Digito              = 0 | 1 | ... | 9.
```

Figura 4.1: Gramática livre de contexto para fórmulas bem formadas do cálculo de predicados

Cálculo de Predicados	Forma Clausal
$\forall x$ Culpado(x)	{ Culpado(x) }
$\exists x$ Culpado(x)	{ Culpado(Alguém) }
$\forall x$ (Sobe(x) $\Rightarrow$ Desce(x))	{ $\neg$ Sobe(x) , Desce(x) }
$\exists x$ Reluz(x) $\wedge$ $\neg$ Ouro(x)	{ Reluz(Algo) } { $\neg$ Ouro(Algo) }
$\forall x$ Homem(x) $\Rightarrow$ ($\exists$ y Necessita(x,y) $\wedge$ $\neg$ Pão(y)	{ $\neg$ Homem(x) , Necessita(x,f(x)) } { $\neg$ Homem(x) , $\neg$ Pão(f(x)) }
$\forall x$ Amiga(Maria,x) $\Rightarrow$ Mora(x,Santos)	{ $\neg$ Amiga(Maria,x) , Mora(x,Santos) }

Tabela 4.5: Exemplos de sentenças na forma clausal

de conhecimento Δ, usando a estratégia, são de fato implicações lógicas de Δ, enquanto estratégias completas podem ser utilizadas para derivar todas as possíveis implicações lógicas de Δ.

As estratégias de inferência mais utilizadas em computação são baseadas no principio da resolução. A utilização deste principio como estratégia de inferência foi introduzida em (ROBINSON, 1965), junto com demonstrações de corretude e completude. A ampla adoção das estratégias de inferência por resolução, na área da computação, deve-se ao fato de estas poderem ser facilmente representadas por procedimentos computacionais. Para começar, estas estratégias trabalham sobre um subconjunto simplificado, mas igualmente expressivo, do cálculo de predicados, no qual as sentenças se resumem a disjunções de fórmulas atômicas, negadas ou não, chamadas *cláusulas*. As cláusulas são geralmente representadas como conjuntos: delimitadas por chaves, e com vírgulas no lugar do operador de disjunção. Cada fórmula atômica, ou sua eventual negação, é denominada *literal*. Exemplos de sentenças do cálculo de predicados, escritas na forma de cláusulas (*forma clausal*), podem ser vistas na tabela 4.5.

Uma outra característica interessante das estratégias de resolução, do ponto de vista computacional, é que existe uma única regra de inferência a ser aplicada a cada passo da derivação: a regra da resolução. Esta regra é utilizada para combinar duas cláusulas $\{\Phi, \Psi_1, \Psi_2, ...\}$ e $\{\neg\Phi, \Theta_1, \Theta_2, ...\}$ em uma nova cláusula, denominada *resolvente*, $\{\Psi_1, \Psi_2, ..., \Theta_1, \Theta_2, ...\}$ (Φ e $\neg\Phi$ são eliminadas). As sentenças Φ que aparecem nas duas cláusulas iniciais não precisam ser exatamente iguais para que possam ser eliminadas pela regra de resolução. Na verdade, basta que elas sejam unificáveis (a grosso modo, que exista uma possível substituição de variáveis que as tornem idênticas). Por exemplo, $Mortal(x)$ é unificável com $Mortal(Hemerson)$, pois existe uma variável,

tude (GÖDEL, 1992)

x, no primeiro literal, que poderia ser substituída por $Hemerson$. Já, $Mortal(Maria)$ e $Mortal(Hemerson)$ não são unificáveis. O primeiro algoritmo linear de unificação foi proposto em (PATERSON; WEGMAN, 1978), uma versão simplificada deste algoritmo, proposta inicialmente em (GENESERETH; NILSSON, 1988), será apresentada na seção 4.1.

O núcleo de qualquer estratégia de resolução consiste então em, dadas duas cláusulas, buscar por literais unificáveis que possam ser eliminados (aparecem negados em uma e apenas uma das cláusulas) e acrescentar no conjunto de cláusulas o resolvente. Como as estratégias de resolução são baseados em prova por contradição (a negação da conclusão, e não a conclusão, é inserida na base de conhecimento original), o procedimento termina assim que uma contradição for derivada (na forma clausal uma contradição é um conjunto vazio).

O que distingue basicamente as diferentes estratégias de resolução é o critério de escolha das duas cláusulas a serem resolvidas a cada passo. Algumas destas estratégias, como a resolução-SLD (*Linear resolution with a Selection function for Definite sentences*), utilizada em Prolog (STERLING, 1994), impõe restrições adicionais no formato da cláusulas, comprometendo um pouco a expressividade [2] em favor da eficiência na execução do procedimento. Entre as outras estratégias usuais podemos citar a resolução unitária (*unit resolution*), linear (*linear resolution*), por conjunto de suporte (*set of support resolution*) e ordenada (*ordered resolution*) (GENESERETH; NILSSON, 1988).

4.1 Unificação

O problema da unificação é basicamente um problema de casamento de padrões. Dois literais são ditos unificáveis quando existe um conjunto apropriado de substituições de variáveis que podem tornar os literais idênticos. A unificação é justamente um processo capaz de responder se dois literais são unificáveis e de oferecer o conjunto de substituições que os torna idênticos. Um conjunto de substituições, também chamado simplesmente de *substituição* (GENESERETH; NILSSON, 1988), é um conjunto

[2] Em Prolog, cujo mecanismo de inferência é basicamente um algoritmo de busca em profundidade, temos ainda o problema da ordenação das cláusulas. Diferentes ordenações, embora equivalentes quanto ao significado, podem fazer com que o grafo de busca possua laços, que em última instância, levam a uma situação de *loop* infinito

1	$\{x/A, y/A, z/A, w/A\}$
2	$\{x/A, y/F(B), z/A, w/F(A)\}$
3	$\{x/A, y/B, z/A, w/F(A)\}$
4	$\{x/z, y/B, w/F(A)\}$
5	$\{x/z, y/B, w/F(z)\}$

Tabela 4.6: Exemplos de Substituições

de associações entre variáveis e expressões no qual cada variável é associada a no máximo uma expressão; e além disto, nenhuma variável já associada a alguma expressão pode ocorrer dentro desta mesma expressão (as associações não podem ser auto-recursivas).

A tabela 4.6 mostra algumas substituições possíveis considerando o par de literais $\Phi = P(x, F(A), F(y))$ e $\Psi = P(z, w, F(B))$. É fácil perceber que a substituição 1 não torna os literais idênticos, e o mesmo ocorre com a segunda substituição, uma vez que $F(B) \neq F(F(B))$. As outras substituições unificam Φ e Ψ, no entanto, a substituição 3, quando aplicada aos literais, produz a expressão $\Theta = P(A, F(A), F(B))$, enquanto a substituição 4 produz $\Theta' = P(z, F(A), F(B))$. A expressão Θ' é claramente mais geral que Θ, pois a última é um caso especial da primeira (a variável z é instanciada pela constante A). Expressões mais gerais são mais úteis em provas de teorema pois permitem uma capacidade maior de geração de novas expressões, por isso, algoritmos de unificação devem ser projetados para retornar sempre o unificador mais geral (UMG) [3]. Exemplo de conjuntos de associações que não são substituições, pois desrespeitam a regra da associação auto-recursiva, são $\{x/F(x)\}$ e $\{x/F(y), y/F(x)\}$.

O algoritmo 4.1 representa um procedimento recursivo capaz de computar o unificador mais geral de duas expressões literais, retornando o valor $falso$ quando estas não são unificáveis. No algoritmo, expressões são representadas através de listas: por exemplo, a expressão $P(x, F(A, B), G(y), A)$ é representada pela lista de dois níveis $< P, x, < F, A, B >, < G, y >, A >$. Duas funções auxiliares são referenciadas no algoritmo: a função ElementoEm(x,i) retorna o i-ésimo elemento da lista x e o procedimento *VerificaRecursão*, usado para eliminar associações auto-recursivas, é apresentado pelo algoritmo 4.2.

[3] A demonstração de que o unificador mais geral é único, a menos de isomorfismos no nome das variáveis, pode ser encontrada em (PATERSON; WEGMAN, 1978)

Algoritmo 4.1 UMG

entrada: Duas expressões Φ e Ψ
saída: Unificador mais geral α, ou o valor $falso$

```
1: se Φ = Ψ então
2:    Retorna()
3: senão se Φ é uma variável então
4:    Retorna(VerificaRecursão(Φ,Ψ))
5: senão se Ψ é uma variável então
6:    Retorna(VerificaRecursão(Ψ,Φ))
7: senão se Φ ou Ψ é constante então
8:    Retorna(falso)
9: senão se Φ e Ψ tem tamanho diferente então
10:    Retorna(falso)
11: senão
12:    α ← Ø
13:    para i = 1 até o tamanho de Φ faça
14:       β ← UMG(ElementoEm(Φ,i),ElementoEm(Ψ,i))
15:       se β = falso então
16:          Retorna(falso)
17:       senão
18:          α ← α ∪ β
19:          Aplica substituição α à Φ e Ψ
20:       fim se
21:    fim para
22:    Retorna(α)
23: fim se
```

Algoritmo 4.2 VerificaRecursão

entrada: Duas expressões x e y
saída: Uma associação x/y ou o valor $falso$

```
1: se x ocorre em y então
2:    Retorna(falso)
3: senão
4:    Retorna(x/y)
5: fim se
```

Sexo(João,Masculino)	Profissão(João,Médico)	Pai(João,Joaquim)
Sexo(Maria,Feminino)	Profissão(Maria,Médico)	Pai(Maria,Joaquim)
Sexo(Pedro,Masculino)	Profissão(Pedro,Professor)	Pai(Pedro,Osório)
Sexo(Carlos,Masculino)	Profissão(Joaquim,Astrônomo)	...
Sexo(Joaquim,Masculino)	...	
Sexo(Osório,Masculino)		
Sexo(Nilza,Feminino)		
Sexo(Raquel,Feminino)		
Sexo(Penelope,Feminino)		
...		

Tabela 4.7: Exemplo de base de conhecimento para SSR

4.2 Satisfação Seqüencial de Restrições

A Satisfação Seqüencial de Restrições (SSR) é uma especialização do cálculo de predicados, na qual a base de conhecimentos é composta unicamente por literais positivos completamente instanciados (sem variáveis). Além disto, as questões formuladas sobre essa base de conhecimentos consistem apenas em conjunções de literais (GENESERETH; NILSSON, 1988). Com estas restrições, a prova de teoremas em SSR pode ser executada de maneira eficiente (polinomial) utilizando a técnica de resolução ordenada (SMITH, 1985; LEDENIOV; MARKOVITCH, 1998), que tem como base o algoritmo de unificação.

A tabela 4.7 mostra um exemplo de uma base de conhecimento de um problema de SSR. Uma possível consulta a esta base de conhecimentos, em que se procura saber quais são as mulheres cujo pai é astrônomo, seria:

$$\text{Sexo}(x,\text{Feminino}) \wedge \text{Pai}(x,y) \wedge \text{Profissão}(y,\text{Astrônomo})$$

A prova de teoremas em SSR se reduz a uma busca de literais unificáveis, na base de conhecimento, para cada uma das expressões (também literais) que compõem a consulta. Em geral, os literais da consulta são unificados da esquerda para direita. A ordem dos literais na consulta pode ter um impacto significativo no desempenho das buscas. No exemplo específico, primeiramente seriam encontradas todas as mulheres da base de conhecimento, depois todos os pais destas mulheres e por último determinar-se-ia quais dentre estes pais são astrônomos. Se invertêssemos a ordem, encontraríamos primeiramente todos os astrônomos, depois os filhos destes astrônomos e, dentre esses

filhos, as mulheres. Se a base de conhecimentos representasse o senso brasileiro, por exemplo, certamente teríamos um ganho significativo em desempenho fazendo a consulta na ordem inversa (uma vez que o número de astrônomos no Brasil é muito menor que o número de mulheres). Consideraremos novamente a questão do desempenho no capítulo 6, já no contexto dos dispositivos adaptativos.

Parte II

Resultados Teóricos

5 ÁRVORES DE DECISÃO ADAPTATIVAS

Apresentaremos neste capítulo um novo enfoque para a indução de árvores de decisão, baseada na teoria dos dispositivos adaptativos, que pode ser aplicada a diversos problemas da área de aprendizagem computacional. Este enfoque combina as características discretas de um dispositivo adaptativo, cujo mecanismo subjacente é uma árvore de decisão, com estratégias para tratamento de valores contínuos, incorretos e inconsistentes.

Iniciaremos descrevendo o mecanismo subjacente a ser utilizado na formalização das árvores de decisão adaptativas. Este novo mecanismo, que será denominado *árvores de decisão não-determinísticas*, generaliza o conceito de árvore de decisão apresentado na seção 3.1. A versão adaptativa deste mecanismo é apresentada na seção 5.1, juntamente com alguns exemplos ilustrativos. As extensões do mecanismo que possibilitam o tratamento de valores contínuos, ausentes ou inconsistentes, são mostradas na seção 5.3.2, que é seguida por uma seção de cálculo de complexidade no tempo e espaço. Finalmente, apresentaremos alguns resultados experimentais relacionados com o desempenho de nossa proposta em problemas de aprendizagem de máquina, bem como conclusões e propostas para trabalhos futuros.

5.1 Árvores de Decisão Não-Determinísticas

Uma árvore de decisão não-determinística é uma árvore de decisão (QUINLAN, 1996) que incorpora o conceito de não-determinismo, bastante estudado na teoria dos autômatos. A introdução do não-determinismo possibilita um tratamento elegante de problemas relacionados com informação incompleta ou ausente, além de propiciar uma distinção conceitual entre decisões exatas e decisões aproximadas.

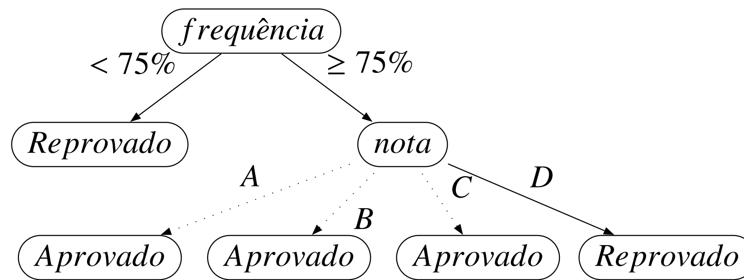


Figura 5.1: Exemplo de uma árvore de decisão

Em uma árvore de decisão convencional, uma decisão só é obtida quando houver algum caminho ligando a raiz da árvore a alguma de suas folhas. A ocorrência de valores inconsistentes ou a não-disponibilidade de valores para certos atributos, durante a tentativa de localizar esse caminho, causa a interrupção do processo, sem que qualquer resposta seja obtida. Com as árvores de decisão não-determinísticas, a busca de um caminho que liga a raiz da árvore a alguma de suas folhas pode continuar, indeterministicamente, usando todas as arestas que partem do nó representando o atributo com valores ausentes ou inconsistentes. Quando trabalhando em modo não-determinístico, a árvore de decisão pode fornecer mais de uma resposta para a mesma questão. Neste caso, a resposta final pode ser uma distribuição probabilística estimada através das ocorrências das diferentes respostas [1].

A figura 5.1 mostra uma árvore de decisão para determinar se um aluno deve ser aprovado ou reprovado, com base na sua freqüência e nota. Nessa árvore, se a freqüência de um determinado aluno fosse desconhecida e sua nota fosse D, a definição original de árvore de decisão não permitiria a obtenção de uma classificação, pois não seria possível escolher uma aresta saindo da raiz. Com as árvores não-determinísticas encontraríamos dois caminhos ligando a raiz às folhas (linhas preenchidas na figura 5.1), ambos apontando uma reprovação. Já para o caso de um aluno com nota C, também com freqüência desconhecida, a árvore de decisão não-determinística retornaria algo como: $p(reprovado) = 0.5, p(aprovado) = 0.5$, em que p indica a probabilidade de ocorrência de um evento (duas folhas encontradas - uma contendo o valor *reprovado* e outra contendo o valor *aprovado*).

Uma árvore de decisão não-determinística, ou árvore-DND, é formalmente defi-

[1]Existem algoritmos de indução de árvores de decisão que trabalham de maneira similar, no entanto, uma das principais contribuições de nossa proposta está na maneira em que os conceitos foram formalizados

nida como uma 7-upla $T = (I, \Sigma, \Gamma, f, c, A, R)$ na qual

I, Σ, Γ são conjuntos não-vazios representando, respectivamente: *exemplos*, *atributos* e *valores*. Tanto Γ quanto Σ são conjuntos finitos, enquanto I é contavelmente infinito. O conjunto Γ contém o símbolo "**?**", denominado *valor ausente* que deve ser usado para representar valores ausentes, desconhecidos ou inexistentes.

$f : \Sigma \rightarrow 2^{\Gamma}$ é uma função que mapeia cada atributo em Σ em um conjunto de valores possíveis para tal atributo. Ou seja, f determina o domínio de cada atributo.

$c \in \Sigma$ define o atributo classe, cujos valores são mostrados nas folhas de uma árvore de decisão.

$A : I \times \Sigma \rightarrow \Gamma$ é uma função binária usada para descrever os elementos do conjunto de exemplos, criando uma associação entre exemplos, atributos e valores. Os *exemplos de treinamento*, usados na fase de construção da árvore de decisão, são caracterizados por possuírem um valor ausente como atributo classe, ou seja, i é um exemplo de treinamento se e somente se $A(i, c) = \textbf{?}$. Quando $A(i, c) \neq \textbf{?}$, temos que i é um *exemplo de teste*. Impomos ainda uma restrição adicional sobre A para evitar que atributos sejam associados a valores que não pertençam ao seu domínio: $A(i, \sigma) \in f(\sigma)$ para todo $i \in I$ e $\sigma \in \Sigma$.

R é uma estrutura hierárquica finita, denominada *sub-árvore*, e definida recursivamente como:

Folha Uma dupla (id, v), na qual $v \in f(c)$ é um valor para o atributo classe, e id é um identificador único ou

Não Folha Uma (n+2)-upla $(id, a, (v_1, R_1), ..., (v_n, R_n))$, na qual id é um identificador, $a \in \Sigma - \{c\}$ é um atributo, $v_i \in f(a), 1 \leq i \leq n$ são valores e todos os elementos $R_i, 1 \leq i \leq n$ são sub-árvores.

O topo da hierarquia induzida por R é denominado *raiz* da árvore de decisão. Os termos *filho* e *pai* serão utilizados da maneira usual, em referência às sub-árvores que se encontram em posição imediatamente inferior ou superior na hierarquia induzida por R. Para formalizar o modo de execução de uma árvore-DND basear-nos-emos na proposta de dispositivo não-adaptativo descrita na seção 2.2. Primeiramente, definimos o espaço de configurações $\mathscr{C}$ de uma árvore-DND como sendo um conjunto de

Algoritmo 5.1 Operação da árvore-DND ao atingir uma configuração final

1: **se** Exemplo de teste **então** $\{A(\iota, a) = \mathbf{?}\}$
2: **se** árvore-DND em modo de operação determinístico **então**
3: acrescente v na saída do dispositivo
4: **senão**
5: acrescente ? e v na saída do dispositivo
6: **fim se**
7: **senão**
8: nada é feito pois uma árvore-DND não é capaz de aprender
9: **fim se**
10: **se** existe um exemplo i disponível na entrada **então**
11: coloque o dispositivo na configuração $(raiz, i)$ e continue a operação
12: **senão**
13: aborte a operação do dispositivo
14: **fim se**

pares ordenados (τ, ι), no qual $\tau \in R$ é uma sub-árvore e $\iota \in I$ é um exemplo. Uma configuração inicial, $c_0 \in \mathscr{C}$, é caracterizada por um par ordenado formado pela *raiz* da árvore e uma instância qualquer $\iota \in I$. Configurações finais são todas aquelas em que o primeiro elemento do par ordenado é uma folha. A seqüência de estímulos de entrada é constituída de exemplos (elementos do conjunto I), enquanto a saída do dispositivo é uma seqüência de elementos de $f(c)$ (classes), que podem ser precedidos pelo símbolo especial "*?*", usado para identificar uma saída não-determinística. A operação de uma árvore-DND prossegue enquanto houver exemplos na entrada, conforme o seguinte procedimento:

1. Dada uma configuração não-final (τ', ι), com $\tau' = (id, a, (v_1, R_1), ..., (v_n, R_n))$, T *atinge em um passo* a configuração (τ'', ι), ou, usando o símbolo tradicional para a relação de passo, $(\tau', \iota) \vdash (\tau'', \iota)$, se e somente se uma das seguintes condições ocorrem:

   Passo Determinístico: $A(\iota, a) = v_i$ e $\tau'' = R_i$, para algum $1 \leq i \leq n$.

   Passo Não Determinístico do Tipo 1: $A(\iota, a) = \mathbf{?}$ e $\tau'' = R_i$, para $1 \leq i \leq n$. Este passo trata da ocorrência de valores ausentes, determinando que todas as sub-árvores filhas devem ser examinadas "paralelamente".

   Passo Não Determinístico do Tipo 2: $\neg\exists i | A(\iota, a) = v_i$, para $1 \leq i \leq n$ e $\tau'' = R_j$, para $1 \leq j \leq n$. Aqui tratamos, de maneira similar ao tipo 1, da ocorrência de um valor para o qual não existe um caminho que possibilite a continuação determinística da busca de uma folha.

2. Quando uma configuração final é atingida, $((id, v), \iota)$, $v \in f(c)$, a operação da árvore-DND segue o algoritmo 5.1

Uma árvore-DND é projetada para funcionar apenas como um transdutor, não como um reconhecedor de linguagens. Portanto, os conceitos de configuração de aceitação e rejeição, da definição de um dispositivo adaptativo, são irrelevantes neste caso.

5.1.1 Exemplo de Árvore de Decisão Não-Determinística

A árvore de decisão apresentada na figura 5.1 pode ser formalizada como uma árvore-DND $T = (I, \Sigma, \Gamma, f, c, A, R)$ na qual:

- I é um conjunto de exemplos de estudantes e consultas relacionadas a estudantes, e.g. $\{maria, paulo, pedro, consulta1, consulta2\}$.
- $\Sigma = \{freq\ddot{u}\hat{e}ncia, nota, resultado\}$
- $\Gamma = \{MenorQue75, MaiorOuIgual75, A, B, C, D, Aprovado, Reprovado, \mathbf{?}\}$
- f é definida através da seguinte tabela:

Σ	f
$freq\ddot{u}\hat{e}ncia$	$\{MenorQue75, MaiorOuIgual75, \mathbf{?}\}$
$nota$	$\{A, B, C, D, \mathbf{?}\}$
$resultado$	$\{Aprovado, Reprovado, \mathbf{?}\}$

- $c = resultado$
- A é definida pela seguinte tabela (em um formato bastante utilizado em inteligência artificial para representar vetores de atributos):

I	$freq\ddot{u}\hat{e}ncia$	$nota$	$resultado$
$maria$	$MaiorOuIgual75$	A	$Aprovado$
$paulo$	$MaiorOuIgual75$	C	$Aprovado$
$pedro$	$MenorQue75$	A	$Reprovado$
$consulta1$	$\mathbf{?}$	A	$\mathbf{?}$
$consulta2$	$MaiorOuIgual75$	D	$\mathbf{?}$

- R é definida como
  - $(R_0, freqüência, (MenorQue75, R_1), (MaiorOuIgual75, R_2))$
  - $(R_1, Reprovado)$
  - $(R_2, nota, (A, R_3), (B, R_4), (C, R_5), (D, (R_6, Reprovado)))$
  - $(R_3, Aprovado)$
  - $(R_4, Aprovado)$
  - $(R_5, Aprovado)$

Note que na definição de R, o identificador de sub-árvore (id) pode ser utilizado para referenciar toda a sub-árvore. A sub-árvore R_6 foi propositadamente deixada dentro de R_2 para ilustrar que também é possível utilizar a própria descrição da sub-árvore, dentro de outra. A seguinte seqüência de configurações mostra a operação da árvore-DND quando o exemplo $consulta2$ é fornecido como estímulo de entrada para o dispositivo:

$((R_0, freqüência, (MenorQue75, R_1), (MaiorOuIgual75, R_2)), consulta2) \Rightarrow$
$((R_2, nota, (A, R_3)), (B, R_4), (C, R_5)), (D, R_6))), consulta2) \Rightarrow$
$((R_6, Reprovado), consulta2)$ // O valor $Reprovado$ é escrito na saída de T

5.2 Árvores-DND Adaptativas

A aplicação direta da tecnologia adaptativa sobre o mecanismo subjacente apresentado na seção anterior resulta em um novo dispositivo denominado árvore de decisão não-determinística adaptativa, ou árvore-DND adaptativa. A camada adaptativa permite agora que a estrutura hierárquica R seja modificada, antes ou depois de cada mudança de configuração ocorrida no mecanismo subjacente. Todas as adaptações são obtidas através da execução de um conjunto de *ações adaptativas elementares*, que podem consultar, inserir ou remover sub-árvores. As ações adaptativas posteriores e anteriores podem ser conectadas a qualquer sub-árvore.

Seguindo a sugestão em (NETO; PARIENTE, 2002), a sintaxe usada na especificação da camada adaptativa segue de perto a sintaxe do mecanismo subjacente, facilitando assim a utilização dela por pessoas que já tenham familiaridade com o dispositivo subjacente. No caso das árvores-DND adaptativas, ações adaptativas elementares podem ser

```
FunçãoAdapt   = Cabeçalho { AçãoElementar }.
Cabeçalho     = NomeFunção [ "(" Variável { "," Variável } ")" ].
AçãoElementar = TipoAção "[" SubÁrvore "]".
TipoAção      = "?" | "+" | "-".
SubÁrvore     = ( [AcaoAnt] ( NaoFolha | Folha ) [AcaoPos] )
                  | Variavel.
NaoFolha      = "(" SubArvId "," Atributo
                    { "," "(" Valor "," SubÁrvore ")" }.
Folha         = "(" SubArvId "," Classe ")".
AcaoAnt       = AçãoAdap.
AcaoPos       = AçãoAdap.
AçãoAdap      = "[" NomeFunção
                    [ "(" Identif { "," Identif } ")" ]
                "]".
SubArvId      = Identif | Variável | Gerador.
Atributo      = Identif | Variável.
Valor         = Identif | Variável.
Classe        = Identif | Variável.
NomeFunção    = Identif.
Variável      = "?" Identif.
Gerador       = "*" Identif.
Identif       = Letra { Letra | Dígito | "_" }.
```

Figura 5.2: Gramática para funções adaptativas em árvores-DND adaptativas

representadas por *sub-árvores genéricas*, cercadas por parênteses, e precedidas pelos símbolos que indicam o tipo da ação elementar: “?” para consulta, “+” para inserção e “-” para remoção. Uma *sub-árvore genérica* é uma ênupla, conforme definido na seção 5.1, cujos elementos podem ser substituídos por variáveis e geradores. Variáveis e geradores são implicitamente declarados, sendo denotados, respectivamente, pelos prefixos “?” e “*”. A gramática livre-de-contexto, em notação de Wirth, apresentada na figura 5.2, define parcialmente a sintaxe de uma função adaptativa no contexto das árvores-DND adaptativas. Esta notação difere um pouco daquela apresentada na seção 2.2.1, principalmente pela retirada das ações iniciais e finais e da declaração de variáveis e geradores. Uma proposta ainda mais simplificada para funções adaptativas será apresentada no capítulo 6.

Para completar a definição da sintaxe das funções adaptativas precisamos acrescentar ainda alguns casos especiais, que simplificam a especificação da camada adaptativa. O primeiro oferece um pouco mais de flexibilidade às ações elementares de consulta

ao permitir que elas atuem também sobre a função A, usada para associar valores aos atributos de um exemplo. Também reservamos o símbolo especial ι para denotar o exemplo corrente na cadeia de entrada do dispositivo. Com isto, uma ação elementar de consulta como $?[((\iota, ?atributo), ?valor)]$, pode ser utilizada na recuperação dos valores de cada atributo do exemplo que está sendo lido. Para obter valores para um atributo específico, basta substituir a variável $?atributo$ por uma constante, representando o nome de um atributo. Finalmente, para simplificar a representação de algumas funções adaptativas freqüentemente usadas neste trabalho, assumiremos que todas as inserções ocorrem exatamente na sub-árvore que gerou a chamada de função adaptativa (substituindo esta sub-árvore), a menos que um outro ponto de inserção seja explicitamente especificado.

Uma árvore-DND adaptativa é uma dupla $T = (CS_0, CA)$, na qual CS_0, a camada ou mecanismo subjacente, é uma árvore-DND $CS_0 = (I, \Sigma, \Gamma, f, c, A, R)$ com uma pequena alteração na definição da estrutura R, que agora permite o acoplamento de ações adaptativas a cada uma das sub-árvores de R. O mecanismo adaptativo CA é formado pelo conjunto das funções adaptativas acima descritas. A operação de uma árvore-DND adaptativa segue o procedimento definido para o mecanismo subjacente, descrito na seção 5.1, até que uma função adaptativa seja acionada. A execução da função adaptativa segue o método geral definido para dispositivos adaptativos na seção 2.2, com as alterações sugeridas acima. Uma restrição adicional determina que a execução de ações adaptativas sejam inibidas durante a leitura de um exemplo de teste, fazendo assim que apenas exemplos de treinamento possam gerar alterações na árvore. Este efeito poderia também ser obtido a partir de funções adaptativas mais complexas, que verificassem o tipo do exemplo antes da execução de outras ações elementares. No entanto, optamos por utilizar esta restrição adicional novamente para simplificar o trabalho de especificação.

5.2.1 Exemplo ilustrativo 1

Para ilustrar com um exemplo bastante simples a utilização de um árvore-DND adaptativa, estenderemos a árvore-DND não-adaptativa apresentada na seção 5.1.1, que trata da avaliação de alunos. Com esta extensão criamos um tratamento de exceção “embutido” em uma função adaptativa, e que representa o seguinte critério especial de aprovação: alunos com freqüência abaixo de 75%, mas que tenham obtido a nota

máxima, A, deverão ser aprovados (e não reprovados, como sugere a árvore de decisão original). Tratamentos de exceção são bastante comuns em construção de compiladores (um exemplo, mostrando uma alternativa para tratamento de erros em compiladores, é apresentado no capítulo 11). Usando dispositivos adaptativos podemos obter uma modelagem bem interessante para casos de exceção, mantendo o modelo mais geral intacto, até que haja a necessidade de uma alteração para tratamento da exceção. Esta alteração pode ser desfeita logo após sua utilização, fazendo com que o modelo permaneça simples na maior parte do tempo.

A árvore-DND adaptativa que trata o caso de exceção acima exposto é um sistema (CS_0, CA) em que o mecanismo subjacente CS_0 é o mecanismo T, definido na seção 5.1.1, com uma pequena diferença na sub-árvore R_1, de R. Esta diferença está relacionada com o acoplamento de uma ação adaptativa anterior A_1, a R_1, que passa agora a ser: $[A_1]$ $(R_1, reprovado)$. A função adaptativa A_1 é definida na tabela 5.1 e é responsável por substituir a sub-árvore folha que reprova indiscriminadamente alunos com freqüência menor que 75%, por uma sub-árvore mais complexa, que considera também a nota do aluno, aprovando aqueles com nota A. A nova sub-árvore carrega também uma ação adaptativa A_2, responsável por retornar a árvore à sua situação inicial, após o tratamento da exceção e antes que um novo exemplo seja lido da entrada. É importante lembrar que as funções adaptativas A_1 e A_2 ilustram também o caso especial de função adaptativa em que o ponto de inserção na árvore está implicito (e corresponde exatamente à sub-árvore que gerou a chamada da função adaptativa)

1	A_1	Cabeçalho sem parâmetros
2	+[(R_7 , nota , (A , (R_8 , Aprovado) [A_2]) , (B , (R_9 , Reprovado) [A_2]), (C , (R_{10} , Reprovado) [A_2]), (D , (R_{11} , Reprovado) [A_2]))]	Ação Elementar de Inserção
1	A_2	Cabeçalho sem parâmetros
2	+[[A_1] (R_1 , Reprovado)]	Ação Elementar de Inserção

Tabela 5.1: Função adaptativa do exemplo 1

5.2.2 Exemplo Ilustrativo 2

Segue abaixo um exemplo de função adaptativa que ilustra a forma como remoções e inserções podem ser realizadas em diferentes partes de uma árvore de decisão. Neste

exemplo, a variável *r* poderia ser multiplamente instanciada, resultando na remoção de diferentes sub-árvores. A utilização da variável *r* na ação elementar de inserção pode ser entendida como um "ponteiro" para o local em que a nova sub-árvore deve ser inserida na árvore de decisão subjacente. A função adaptativa deste exemplo realiza uma espécie de compressão na árvore, detectando um certo tipo de redundância que ocorre quando todas as folhas filhas de um determinada sub-árvore possuem o mesmo valor.

1	Compressão	
2	?[$(?r$,atr, $((?id_1, ?valor), (?id_2, ?valor)))$]	Procura por sub-árvores com dois níveis cujo segundo nível é composto por duas folhas com o mesmo valor
3	-[$?r$]	Remove as sub-árvores encontradas
4	+[$(?r, ?valor)$]	Insere, em lugar de todas as sub-árvores encontradas, uma sub-árvore mais simples: uma folha contendo o valor de atributo detectado pela ação elementar de consulta.

5.3 AdapTree-E

AdapTree-E (*Adaptive Tree Extended* é um algoritmo de aprendizagem de máquina criado a partir de união de um tipo especial de árvore-DND adaptativa, que é capaz de lidar apenas com valores discretos, com técnicas para tratamento de valores contínuos. No AdapTree-E, a aprendizagem pode ser incremental, com exemplos de treinamento sendo intercalados com exemplos de teste. Diferente da maioria dos algoritmos de indução de árvores de decisão, o AdapTree-E não busca a construção de uma árvore de decisão mínima. O seu funcionamento está mais próximo ao dos algoritmos de aprendizagem baseados em exemplos (*instanced-based learning*), como o k-NN (*k-Nearest Neighbour*) (AHA; KIBLER; ALBERT, 1991). No AdapTree-E, os exemplos

são todos armazenados em uma estrutura similar a uma árvore de prefixos, na qual cada nível representa um dos atributos do domínio da aprendizagem. É a natureza não-determinística do mecanismo subjacente, a árvore-DND, e as técnicas para tratamento de valores contínuos, que capacitam o AdapTree-E a generalizar além dos exemplos.

A árvore-DND adaptativa que compõe o módulo discreto do AdapTree-E é chamada AdapTree. As funções adaptativas da AdapTree impõem uma ordenação completa, pré-determinada, ao conjunto de atributos Σ. É esta ordenação que determina em que nível da árvore de decisão irão ocorrer os diversos atributos. A estrutura R da AdapTree é composta, em sua configuração inicial, de uma única folha, contendo o valor ausente "?" e uma ação adaptativa anterior. À medida que exemplos vão sendo lidos da entrada do dispositivo, a estrutura R vai crescendo, na forma de uma árvore de prefixos. Como cada nível da árvore corresponde a um único atributo, e vice-versa, a altura máxima da árvore de decisão subjacente da AdapTree é $|\Sigma|$.

Formalizamos a AdapTree como sendo uma árvore-DND adaptativa definida pela tupla $T = ((I, \Sigma, \Gamma, f, c, A, R), \Theta)$ na qual: (1) o conjunto de atributos Σ possui uma ordenação arbitrária e portanto, pode ser representado por uma seqüência $a_1, a_2, ..., a_j$, em que $j = |\Sigma|$, (2) o atributo classe é a_j, (3) $R = [A_1]\ (R_1, ?)$ e finalmente, (4) a camada adaptativa Θ é composta de j funções adaptativas, $A_1, A_2, ..., A_j$, uma para cada atributo. As funções adaptativas A_i, $1 \leq i < j$ (todas exceto a última), não possuem parâmetros e contêm uma única ação adaptativa elementar, de inserção, com o seguinte formato:

$$+[(*r_0, a_i, (f_1(a_i), [A_{i+1}](*r_1, ?)), (f_2(a_i), [A_{i+1}](*r_2, ?)), ..., (f_v(a_i), [A_{i+1}](*r_v, ?)))]$$

,

no qual a_i é o i-ésimo atributo de Σ, $f_k(a_i)$ denota o k-ésimo valor do domínio de a_i, para $1 \leq k \leq v$, e v é o tamanho do conjunto de possíveis valores do atributo a_i, ou seja, $(v = |f(a_i)|)$. A última função adaptativa, A_j, mostrada abaixo, é responsável por criar uma folha contendo a classe do exemplo de treinamento que acaba de ser lido.

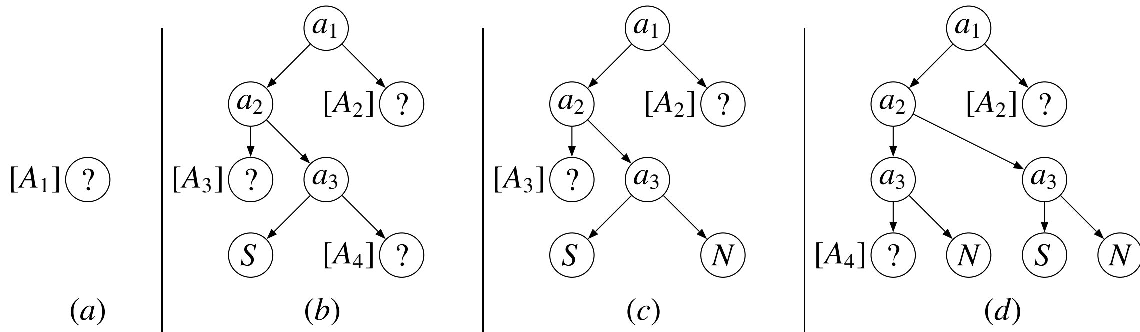


Figura 5.3: Evolução de uma AdapTree
(a) Configuração inicial (b) Após SNSS (c) Após SNNN (d) Após SSNN

1	A_j	
2	$?[((\iota,$nome_do_atributo_classe$), ?\alpha)]$	Obtém o valor do atributo classe
3	$+[(*r, ?\alpha)]$	Adiciona folha (substituindo a sub-árvore que gerou a chamada desta função adaptativa)

A figura 5.3 mostra graficamente a evolução de uma AdapTree para um problema hipotético de aprendizagem com quatro atributos binários (valores sim-não), $a_1, ..., a_4$. O último deles, a_4, corresponde ao atributo classe. Os valores dos atributos de cada exemplo apresentado ao dispositivo são mostrados através de cadeias de S e N, com cada posição da cadeia correspondendo a um dos quatro atributos. Nota-se que todos os exemplos são de treinamento, uma vez que $a_4 \neq ?$, para todos eles. Implicitamente, a aresta que parte do lado esquerdo de um nó corresponde sempre ao valor S (sim), enquanto a do lado direito, ao N (não) - por isto este valores não precisaram ser mostrados no gráfico.

5.3.1 Exemplo Ilustrativo 3

De volta ao exemplo sobre notas e freqüências, uma AdapTree capaz de agir neste domínio pode ser formalizada através da tupla $T = ((I, \Sigma, \Gamma, f, c, A, R), \Theta)$ na qual:

- I,Σ,Γ,f,c e A são os mesmos do exemplo 5.1.1.
- R é formado pela sub-árvore **[**A_1**]** $(R_1, ?)$

1	A_1	Cabeçalho
2	+[($*r_0$, *freqüência*, (MenorQue75, [A_2] ($*r_1$, ?)), (MaiorOuIgual75, [A_2] ($*r_2$, ?)))]	Insere sub-árvore para o atributo freqüência
1	A_2	Cabeçalho
2	+[($*r_0$, *nota* , (A , [A_3] ($*r_1$, ?)), (B , [A_3] ($*r_2$, ?)), (C , [A_3] ($*r_3$, ?)), (D , [A_3] ($*r_4$, ?)))]	Insere sub-árvore para o atributo nota
1	A_3	
2	$?[((\iota, resultado), ?\alpha)]$	Consulta valor do resultado
3	$+(*r, ?\alpha)]$	Insere folha com valor do resultado

Tabela 5.2: Funções adaptativas do exemplo 3

- Θ contém as 3 funções adaptativas, A_1, A_2 e A_3, mostradas na tabela 5.2.

5.3.2 Tratamento de valores contínuos, ausentes e inconsistentes

5.3.2.1 Valores contínuos

O tratamento de valores contínuos no AdapTree-E é feito através da discretização. Na implementação atual, o método de discretização de Fayyad e Irani (FAYYAD; IRANI, 1993) é aplicado, antes do início da operação da árvore-DND adaptativa, para determinar a quantidade e o tamanho dos intervalos que deverão particionar cada um dos atributos contínuos. Este método é do tipo supervisionado, pois necessita de um conjunto de exemplos de treinamento para decidir sobre os pontos de corte que determinarão os intervalos discretos do atributo. Métodos supervisionados têm sido experimentalmente constatados como superiores aos métodos não-supervisionados, quando associados a algoritmos de aprendizagem de máquina e comparados pelo critério de taxa de acerto sobre um conjunto independente de testes (DOUGHERTY; KOHAVI; SAHAMI, 1995; KOHAVI; SAHAMI, 1996). Para explicar o funcionamento do método de discretização empregado no AdapTree-E, iremos primeiramente rever o conceito de entropia e sua relação com ganho de informação.

Dada uma lista S contendo elementos retirados de um conjunto $C = \{c_1, c_2, ..., c_n\}$, diz-se que a entropia de S é um valor numérico que busca medir a homogeneidade da

distribuição dos elementos de C em S. Quando todos os elementos de S são iguais entre si, tem-se, em relação a C, uma distribuição maximalmente heterogênea, pois apenas um dos elementos de C aparece em S, e portanto a entropia deve apresentar o valor mínimo. O valor máximo de entropia é obtido quando todos os elementos de C aparecem em igual quantidade em S. A entropia é inversamente proporcional ao que, em teoria da informação, chamamos de *quantidade de informação* A intuição para esta definição de quantidade de informação pode ser obtida verificando-se que: se sabemos que todos os elementos do conjunto S possuem o mesmo valor c_i (entropia mínima), então podemos determinar, com 100% de possibilidade de acerto, que um elemento retirado aleatoriamente de S terá valor c_i (quantidade máxima de informação). Agora, se a entropia de S é máxima, a probabilidade de acertamos o valor de uma elemento retirado aleatoriamente de S é apenas de $1/|C|$ (50% quando $|C| = 2$ - ou seja, quantidade mínima de informação). Entre estes dois extremos temos toda uma gradação de valores de entropia que dependem da distribuição dos elementos de C em S. Por exemplo, se $C = \{c_1, c_2\}$, temos que a entropia de $S' = \{c_1, c_1, c_1, c_1, c_2\}$ é intuitivamente menor que a de $S'' = \{c_1, c_1, c_2\}$ ou $S''' = \{c_1, c_1, c_1, c_2, c_2\}$. Assumindo que $|S_i|$ seja o total de elementos c_i em S e que $\log(0) = 0$, podemos concretizar a intuição descrita acima, através da fórmula:

$$Entropia(S) = \sum_{1 \leq i \leq n} -\frac{|S_i|}{|S|} \log \frac{|S_i|}{|S|} \tag{5.1}$$

A entropia de um conjunto de listas, $\mathscr{S} = S_1, S_2, ..., S_m$, pode ser definido como a média aritmética das entropias de cada elemento S_j de $\mathscr{S}$, ponderada sobre o tamanho de cada S_j. Ou seja,

$$Entropia(\mathscr{S}) = \sum_{1 \leq j \leq m} \frac{Entropia(S_j)}{|S_j|} \tag{5.2}$$

É possível reduzir a entropia de um conjunto de listas $\mathscr{S}$ repartindo-se os elementos de algumas das listas de $\mathscr{S}$ em listas menores. Para que as listas menores resultantes tenham, sistematicamente, menor entropia que as originais (quando possível), algum tipo de informação sobre os elementos desta lista deve ser utilizado (um particionamento aleatório não é suficiente). Por exemplo, o conjunto de listas $\mathscr{S} = \{< c_1, c_2, c_1, c_2 >\}$, que contém uma única lista com entropia máxima (assumindo apenas 2 tipos de elementos, c_1 e c_2), pode ser transformado no conjunto

$\mathscr{S}' = \{< c_1, c_1 >, < c_2, c_2 >\}$, contendo apenas listas de entropia mínima (ou quantidade de informação máxima). É importante observar que se a listas de $\mathscr{S}$ tivessem sido particionadas de uma outra forma, digamos $\mathscr{S}'' = \{< c_1, c_2 >, < c_1, c_2 >\}$, poderíamos não ter obtido qualquer redução na entropia. Diversos algoritmos de aprendizagem de máquina e também de discretização baseiam-se, direta ou indiretamente, em métodos para redução de entropia (ou ganho de informação). O método de discretização de Fayyad e Irani para um atributo contínuo x consiste em ordenar o conjunto de exemplos de treinamento, usando como chave os valores do atributo x, e buscar um ponto de corte nos valores de x de forma que a entropia dos dois conjuntos resultantes seja a menor possível em relação ao atributo classe. A busca de pontos de corte é efetuada recursivamente sobre os dois conjuntos resultantes, até que um critério de parada seja satisfeito. O critério de parada usado por Fayyad e Irani é baseado no princípio da descrição de comprimento mínimo (MDL - *Minimum Description Length* (RISSANEN, 1978; YU, 1998)).

Realizamos alguns experimentos para comparar o desempenho do AdapTree-E utilizando diferentes métodos de discretização. Os domínios de aplicação foram os mesmos utilizados por Dougherty, Kohavi e Sahamis (DOUGHERTY; KOHAVI; SAHAMI, 1995; KOHAVI; SAHAMI, 1996), que também realizaram comparação de métodos de discretização, mas utilizando os algoritmos C4.5 e naiveBayes para a aprendizagem de máquina. Os métodos comparados foram os de Fayyad e Irani e os métodos não-supervisionados com 3, 5 e 10 intervalos iguais. A tabela 5.3 mostra as taxas de acerto do AdapTree-E quando acoplado aos 4 diferentes métodos de discretização. Também são mostrados na tabela o desvio-padrão e os casos em que os métodos não-supervisionados (3,5 e 10 intervalos) foram estatisticamente superiores (+) ou inferiores (-) ao método de Fayyad e Irani (usando o teste t-Student pareado, com nível de confiança igual a 95%). Os experimentos indicam que, assim como no caso do C4.5 e do naiveBayes, o AdapTree-E apresenta, ao menos em algumas aplicações, uma ligeira melhora de desempenho quando acoplado a um metodo de discretização supervisionado (embora nas bases de dados *anneal*, *crx* e *hepatitis* tenha sofrido perda significativa de desempenho com Fayyad). A implementação do AdapTree-E permite que o método de discretização seja facilmente alterado, o que pode ser interessante em alguns domínios de aplicação (como mostra a tabela 5.3). Um estudo mais aprofundado sobre métodos de discretização pode ser encontrado em (DOUGHERTY; KOHAVI; SAHAMI, 1995). Todas estas bases de dados foram extraidas do amplamente utilizado

Base	Fayyad	AdapTree-E 3-Interv.	5-Interv.	10-Interv.
anneal	94.99±2.17	97.55±1.14+	97.44±1.58+	98±0.88+
breast cancer	71.71±6.9	71.71±6.9	71.71±6.9	71.71±6.9
cleve	78.18±6	73.56±9.5 -	75.89±7	74.89±6.4
crx	80.58±4	83.48±4.39+	83.19±4.59+	81.74±3.63
diabetes	73.05±4.9	72.38±6.24	72.39±5.08	70.57±3.7
german	70.3 ±5.52	70 ±6.06	70.7 ±4.64	70.5 ±4.65
glass	58.83±10.1	66.23±10.07	54.24±7.68	54.65±14.06
heart	78.89±7.21	74.44±9.95	77.41±7.08	77.41±5.08
hepatitis	77.5 ±8.94	74.29±12.6	84.5 ±8.12+	78.04±10.04
horse-colic	68.75±3.84	68.75±3.84	68.75±3.84	68.75±3.84
hypothyroid	98.38±0.71	92.47±0.42-	92.13±0.49-	92.52±0.84-
ionosphere	88.02±5.2	86.6 ±7.63	87.75±4.25	86.34±7.3
iris	92.67±4.92	96 ±3.44+	94 ±6.63	95.33±4.5
sick	97.67±0.65	93.48±0.56-	93.88±0.73-	97.77±1.16
vehicle	65.72±3.99	60.86±5.74-	60.53±4.98	62.39±4.51

Tabela 5.3: AdapTree-E com diferentes métodos de discretização

repositório de aprendizagem de máquina da UCI (BLAKE; MERZ, 1998), disponível na internet [2], de onde podem ser obtidas também descrições detalhadas de cada uma das bases.

5.3.2.2 Valores ausentes

Quinlan (QUINLAN, 1989) aponta que a ausência de valores para alguns atributos de um exemplo, situação bastante comum em aplicações reais, incita três importantes questões relacionadas com algoritmos de aprendizagem de máquina baseados em árvores de decisão: (1) como o número total de valores ausentes deve afetar a comparação entre atributos, (2) como devem ser considerados os valores ausentes se o conjunto de treinamento tiver de ser particionado durante a aprendizagem, e (3) como deve ser tratado um valor ausente durante a fase de teste, quando a árvore de decisão é percorrida da raiz para as folhas. Os dois primeiros problemas, relacionados com a fase de aprendizagem, são tratados no AdapTree-E através de um pré-processamento do conjunto de treinamento, para substituir valores ausentes por valores válidos. Esta substituição, que segue o método usado no algoritmo CN2 (CLARK; NIBLETT, 1989), consiste, para o caso de atributos discretos, em usar no lugar do valor ausente o valor

[2]http://www.ics.uci.edu/ mlearn/MLRepository.html

mais comum (moda estatística, calculada sobre um conjunto de treinamento). Para atributos contínuos utilizamos a média aritmética. O terceiro problema apontado por Quinlan é tratado pelo mecanismo subjacente da AdapTree, conforme explicado na seção 5.1.

5.3.2.3 Valores inconsistentes

Chamamos de exemplos de treinamento inconsistentes, dois exemplos que apresentam o mesmo valor para todos atributos, com exceção do atributo classe. Exemplos deste tipo são comuns em problemas reais em que sensores são utilizados para capturar informação em um ambiente sujeito a diversos tipos de ruído. Prevendo este tipo de problema, o AdapTree-E mantém alguns contadores atrelados a cada uma de suas folhas. Esses contadores permitem que o valor de classe associado a cada folha seja sempre aquele de maior freqüência entre os exemplos inconsistentes que eventualmente são detectados no conjunto de treinamento.

5.3.3 Análise de Complexidade no Tempo e no Espaço

Analisaremos primeiramente a complexidade no espaço da componente discreta do AdapTree-E, uma AdapTree $T = ((I, \Sigma, \Gamma, f, c, A, R), \Theta)$. Em uma AdapTree os atributos não podem ser repetidos em diferentes níveis da árvore de decisão, ou seja, a árvore de decisão induzida possui uma altura máxima igual ao número total de atributos, $m = |\Sigma|$, de T. Além disto, a ramificação de cada nó da árvore é limitada superiormente pelo número máximo de valores que um atributo $a_i \in \Sigma$ pode assumir, que é justamente $|f(a_i)|$, denotado a partir de agora por v_{a_i}. Portanto, o tamanho máximo de uma AdapTree, em números de nós da árvore de decisão, pode ser descrito pela fórmula abaixo (que não depende do tamanho do conjunto de exemplos de treinamento)

$$1 + \prod_{i=1}^{1} v_{a_i} + \prod_{i=1}^{2} v_{a_i} + ... + \prod_{i=1}^{m} v_{a_i} = 1 + \sum_{j=1}^{m} \prod_{i=1}^{j} v_{a_i} \tag{5.3}$$

Fazendo $p = max(v_{a_i}), a_i \in \Sigma$, obtemos uma complexidade, no pior caso, para o espaço ocupado por uma AdapTree, na ordem de $O(p^m)$. Embora seja um resultado exponencial, ele não compromete a utilização prática do AdapTree-E, uma vez

Base	p	m	n	Limite Superior $O(p^m)$	Número de Nós Reais	Memória Física em MBytes
anneal	10	38	898	10^{38}	15207	11
audiology	6	69	226	10^{53}	11897	11
autos	22	25	205	10^{33}	3174	3
breast	13	9	286	10^{10}	1206	1.5
cleve	4	13	303	10^{8}	1606	1.8
diabetes	4	8	768	10^{4}	500	1.3
german	11	20	1000	10^{20}	14327	9.4
glass	4	9	214	10^{5}	302	0.9
hypothyroid	6	29	3163	10^{22}	18595	16
mushroom	12	22	8124	10^{23}	53612	38

Tabela 5.4: Experimentos sobre o custo em espaço do AdapTree-E

que na maioria dos casos, os valores p e m são tão menores que o tamanho do conjunto de treinamento (n), que podem ser considerados constantes. Além disto, com o domínio de aplicação do AdapTree-E definido, temos que p e m podem efetivamente ser considerados constantes em relação a n (que pode variar em diferentes execuções do AdapTree-E). Portanto, em muitos casos, a complexidade em espaço pode ser considerada constante, visto que n não influi no tamanho da árvore de decisão induzida pelo AdapTree. A tabela 5.4 compara o espaço ocupado, projetado pela função exponencial descrita acima, com o número efetivo de nós da árvore decisão quando o AdapTree-E é aplicado a alguns problemas do repositório da UCI. A tabela mostra também o espaço físico (*heap* da máquina virtual Java reservado para a execução de uma instância do AdapTree-E) gasto pelo AdapTree-E, para os mesmos problemas. É fácil perceber por esta tabela que a fórmula obtida é excessivamente pessimista.

Para completar a análise de complexidade no espaço do AdapTree-E, precisamos ainda considerar os módulos de tratamento de exemplos inconsistentes e de discretização. Para tratar valores inconsistentes tivemos de incluir na árvore de decisão alguns contadores, um para cada possível valor de classe, em cada uma das folhas da árvore. É como se tivéssemos aumentando em um nível a árvore de decisão. Se considerarmos que m já inclui o atributo classe, teremos que a fórmula $O(p^m)$ continua sendo válida. A substituição de valores ausentes e a discretização podem ser executadas em um espaço linear em relação aos valores p,m e n, pois envolvem apenas cálculos de moda, média e ordenações.

A complexidade no tempo será analisada, como é usual em aprendizagem de

máquina, separadamente, para as fases de treinamento e aprendizagem, e sobre o tamanho do conjunto de treinamento. A fase de treinamento pode subseqüentemente ser divida nos seguintes passos: discretização, substituição de valores ausentes e construção da árvore de decisão. O custo do método de discretização de Fayyad e Irani é dominado por uma operação de ordenação do conjunto de exemplos, sendo, no pior caso, igual a $O(nlog(n))$ (YANG; WEBB, 2001). O cálculo de modas e médias do passo seguinte é um problema trivial em computação, que pode ser resolvido em $O(n)$. Finalmente, cada exemplo de treinamento é absorvido pela AdapTree por um procedimento de descida na árvore de decisão, que começa na raiz e prossegue até encontrar uma folha. Nesta descida poderão ser acionadas funções adaptativas, que irão construir os nós necessários caso um caminho até alguma folha não exista. Este procedimento, de absorção de um único exemplo, possui um custo constante em relação ao n, e portanto, o custo da absorção de todos os n exemplos do conjunto de treinamento é da ordem de $O(n)$. Temos então que o custo no pior caso do AdapTree-E é dominado pelo custo da discretização, sendo portanto da ordem de $O(nlog(n))$.

Na fase de teste temos basicamente uma descida pela árvore de decisão, da raiz até as folhas. que pode ser efetuada em tempo proporcional à altura da árvore, m (total de atributos), caso o exemplo de teste não possua valores ausentes ou valores para os quais não exista um caminho de descida disponível. Para estes casos, a árvore-DND subjacente à AdapTree irá utilizar o modo de operação não-determinístico, que exige um cálculo de custo um pouco mais sofisticado. Chamaremos de *atributo problemático* os atributo cujos valores, no exemplo que está sendo classificado, geram uma execução não-determinística. Sempre que um atributo problemático a é encontrado durante a descida da árvore de decisão, temos que, no máximo v_a (valores possíveis para o atributo) novos caminhos passam a ser percorridos. No pior caso, em que todos os atributos do exemplo são problemáticos, temos que todos os nós da árvore acabam sendo percorridos, elevando o custo para $O(p^m)$ (tamanho máximo da árvore de decisão). De qualquer forma, o custo para classificação de um exemplo continua sendo constante em função de n. A figura 5.4 ilustra os caminhos (linhas sólidas) percorridos em uma árvore de decisão para um exemplo com os atributos a_2 e a_4 problemáticos.

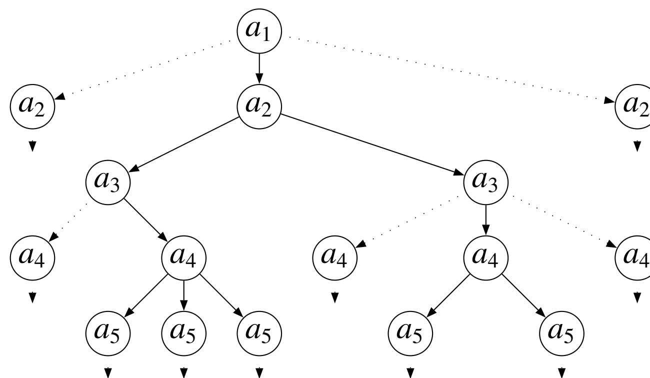


Figura 5.4: Árvore-DND classificando exemplo com valores ausentes

5.3.4 Experimentos

O AdapTree-E foi implementado em Java e integrado ao pacote WEKA [3] (*Waikato Environment for Knowledge Analysis*) (WITTEN; FRANK, 2000). O WEKA é uma excelente ferramenta para comparação de diferentes algoritmos de aprendizagem de máquina. Além de oferecer implementações dos mais conhecidos algoritmos de aprendizagem de máquina, possui um módulo estatístico bastante poderoso e versátil, capaz de gerar automaticamente diversos tipos de tabelas comparativas. A tabela 5.5, gerada pelo WEKA, compara a taxa de acerto do AdapTree-E com aquelas obtidas pelos algoritmos *naiveBayes* (DOMINGOS; PAZZANI, 1996), 3-NN (MITCHELL, 1997) (*k-Nearest Neighbourhood* com k = 3), C4.5 (QUINLAN, 1996) e Id3 (QUINLAN, 1996). Alguns destes algoritmos podem ter seu desempenho modificado através de ajuste de parâmetros. Neste experimento optamos pela utilização dos valores padrão fornecidos pelo WEKA, como ocorre geralmente em experimentos realizados através do WEKA e relatados na literatura da área. A taxa de acerto (e o valores de desvio padrão) mostrada na tabela é a média obtida após 10 execuções de cada algoritmo sobre cada uma das bases de teste. A técnica utilizada na separação do conjunto de exemplos entre treinamento e teste, denominada *validação cruzada estratificada de 10 dobras* [4], consiste basicamente em dividir o conjunto de exemplos em 10 partes, usando, a cada execução, uma parte diferente para teste e as 9 restantes para treinamento. Os sinais de mais e menos apontam desempenho significativamente (estatística t-Student com confiança igual a 95%) superiores e inferiores, respectivamente, sempre em relação ao AdapTree-E (por isto a coluna do AdapTree-E não apresenta nenhum sinal de mais ou

[3] Software livre e com fontes abertas disponível em http://www.cs.waikato.ac.nz/~ml/weka/

[4] *10-Fold Stratified Cross-Validation*

Base	AdapTree-E	naiveBayes	KNN	C4.5	Id3
anneal	94.99±2.17	86.53±3.45 -	97.44±1.9 +	98.44±0.78 +	99.67±0.75 +
audiol.	69.35±7.17	65.58±5.38 -	53.12±7.85 -	74.03±5.37	77.91±5.3 +
autos	75.43±6.02	56.57±4.91 -	55 ±4.22 -	70.86±7.6 -	80.02±8.8 +
breast	71.71±6.9	74.06±9.83	73.42±6.77	75.18±6.37	57.67±7.71 -
cleve	78.18±6	84.46±6.69 +	82.49±7.5	79.17±6.75	62.99±9.94 -
diabetes	73.05±4.9	75.78±3.61	74.74±5.73	74.09±4.55	60.93±4.85 -
german	70.3±5.52	74.9±3.9 +	74.2±3.43 +	69.7±4.47	60.4±5.52 -
glass	58.83±10.1	48.48±6.34 -	70.11±5.35 +	67.16±10.08 +	51.9±10.56 -
heart	78.89±7.21	85.19±7.2 +	78.89±8.74	77.78±8.73	62.96±9.56 -
hepatitis	77.5±8.94	83.79±7.08	82.54±7.44	79.42±7.91	73±8 -
horse	68.75±3.84	66.59±5.34	66.04±7.75	66.31±1.29	10.33±3.35 -
hypothy.	98.14±0.37	95.2±0.53 -	93.22±0.32 -	99.56±0.15 +	90.4±0.69 -
ionosph.	88.02±5.2	82.36±6.04 -	85.18±5.19	90.9±5.31	83.78±6.67 -
iris	92.67±4.92	96 ±4.66	95.33±5.49	95.33±4.5 +	92±6.13
lymphog.	75.4±5.08	80.8±4.34 +	82.6±3.78 +	75.8±6.21	75±12.97
mushro.	100 ±0	95.38±0.37 -	99.96±0.06	100 ±0	100±0
sick	97.67±0.65	92.74±1.57 -	96.21±0.57 -	98.65±0.54 +	97.77±0.61
vehicle	65.72±3.99	44.45±4.38 -	69.15±5.2	73.39±4.43 +	62.17±4.49
vote	93.92±1.72	90.2±1.02 -	93.24±1.42	95.88±1.25 +	93.56±3.21
vowel	74.78±2.44	62.11±3.43 -	76.65±2.27	76.44±0.89	78.99±4.66 +

Tabela 5.5: Taxas de acerto para AdapTree-E, naiveBayes, KNN, C4.5 e Id3

menos). Novamente, as bases de dados foram extraídas do repositório da UCI. A tabela 5.6 apresenta algumas caracteristicas importantes destas bases, como a quantidade de atributos contínuos, o total de exemplos e o percentual de valores ausentes.

Estes experimentos indicam que o AdapTree-E oferece um desempenho comparável, nos diversos domínios de aplicação representados pelas bases de treinamento utilizadas, ao de algoritmos já consolidados na área de aprendizagem de máquina. Em 16 das 20 bases de dados, o AdapTree-E apresentou igual (5 casos) ou melhor (11 casos) resultado que o *naiveBayes*. A comparação com o kNN mostrou-se bastante equilibrada, com empates em 12 das bases de dados, e 4 vitórias para cada um dos algoritmos nas bases restantes. O classificador C4.5 mostrou melhor desempenho em 7 bases, mas embora o AdapTree-E tenha superado o C4.5 em apenas uma das bases, mostrou-se competitivo em outras 12. É importante destacar que o desempenho do AdapTree-E foi muito bom nas bases contendo apenas atributos discretos (que são tratados diretamente pela árvore-DND adaptativa embutida no AdapTree-E): *audiology*,*breast*, *mushroom* e *vote*, o que ajuda a rebater eventuais argumentos que relacionem o bom desempenho do AdapTree-E apenas ao módulo de discretização.

Bases	Atributos Contínuos	Atributos Discretos	Total de Exemplos	Percentual de Valores-Ausentes
anneal	6	32	898	62%
audiology	0	69	226	0.06%
autos	15	10	205	1%
breast	0	9	286	0.3%
cleve	6	7	303	0.2%
diabetes	8	0	768	0%
german	7	13	1000	0%
glass	9	0	214	0%
heart	13	0	270	0%
hepatitis	6	13	150	5%
horse-colic	7	20	368	18.7%
hypothyroid	7	22	3163	5.4%
ionosphere	34	0	341	0%
iris	4	0	150	0%
lymphography	3	15	148	0%
mushroom	0	22	8124	1%
sick	7	22	3772	5.4%
vehicle	18	0	946	0%
vote	0	16	435	3%
vowel	10	3	990	0%

Tabela 5.6: Descrição dos conjuntos de dados usados no experimento com AdapTree-E

A comparação com o Id3 é especialmente importante pois, como o AdapTree-E, ele é o único dos algoritmos comparados que também depende de discretização prévia dos atributos. Utilizando o mesmo método de discretização, de Fayyad e Irani, tanto para o Id3 quanto para o AdapTree-E, podemos perceber pela tabela 5.5 que o nosso algoritmo apresenta desempenho significativamente superior em 10 das bases de dados, contra apenas 4 em que o Id3 é superior.

5.4 Consideração sobre Ordenação dos Atributos

Em geral, o AdapTree-E utiliza uma ordenação arbitrária, para os atributos, na geração da árvore de decisão. Nos testes apresentados na seção anterior, por exemplo, utilizamos a ordem em que os atributos são descritos nos arquivos de teste disponíveis na UCI. Não é difícil perceber que esta ordenação não tem qualquer influência sobre o desempenho do algoritmo em relação a taxa de acerto. Isto acontece, primeiro, porque a folha gerada pela leitura de um determinado exemplo de treinamento existirá em todas as possíveis árvores induzidas pelo AdapTree-E, independemente da ordem dos atributos (a definição das funções adaptativas no AdapTree-E assegura esta propriedade). E segundo, porque, na ocorrência de atributos problemáticos durante a classificação de novos exemplos, todas as alternativas de descida na árvore, a partir deste atributo, são consideradas; ou seja, a amostra sobre a qual o AdapTree-E irá definir a classificação de exemplo de teste, no caso de indeterminismos, é sempre a mesma (independentemente da ordem dos atributos). A classificação de exemplos sem atributos problemáticos também é, obviamente, independente da ordem dos atributos.

A figura 5.5 mostra, genericamente, o percurso percurrido durante a classificação de um determinado exemplo, em uma AdapTree-E com n atributos, $a_1, a_2, ..., a_n$; e um atributo problemático, a_i, com $k \geq 2$ possibilidades de descida. Embora a ordem dos atributos não seja importante em relação à taxa de acerto, percebemos pela figura 5.5 que esta ordem pode ter impacto no tempo de execução do algoritmo, na fase de classificação de exemplos. Quanto mais próximos das folhas estiverem os atributos problemáticos, menos nós terão que ser visitados durante a classificação de um exemplo. De fato, a quantidade de nós visitados nesta situação é igual a $i + (k * (n - i))$, com $k \geq 2$ e $0 \leq i < n$ (extritamente menor que n porque o atributo problemático não pode ser o atributo classe).

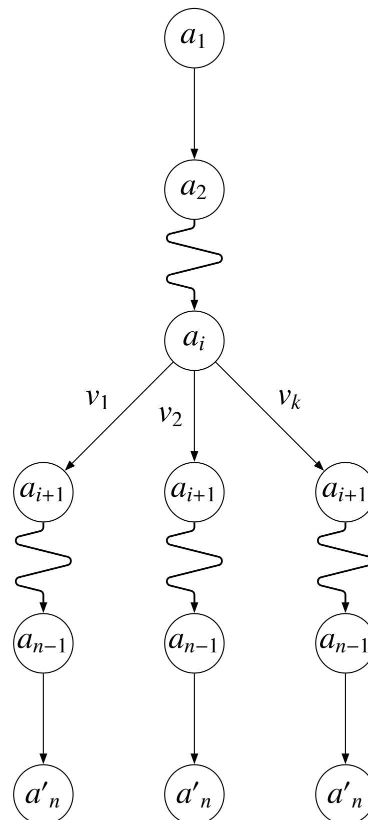


Figura 5.5: Impacto da ordem dos atributos no desempenho de uma AdapTree-E

Agora, como foi destacado na seção 5.3.3, o desempenho do AdapTree-E, na fase de classificação, é ainda independente da quantidade de exemplos de treinamento. A tabela 5.7 ilustra o desempenho comparativo do AdapTree-E, em relação a outros algoritmos de aprendizagem de máquina (estes resultados também foram obtidos através do WEKA), para algumas das bases de dados da UCI.

Dataset	AdapTree	Id3	NBayes	5NN	C4.5	Backpro
iris	0s	0s	0.01s	0.05s	0.01s	0.01s
ionosphere	0.01s	0.01s	0.04s	1.3s	0.01s	0.13s
hypothyroid	1.3s	0.11s	0.45s	184.48s	0.07s	1.17s
hepatitis	0s	0s	0.01s	0.18s	0s	0.02s
Glass	0.01s	0s	0.02s	0.37s	0.01s	0.01s

Tabela 5.7: Comparação de Tempo para Classificação

5.5 Conclusões

Um novo algoritmo de aprendizagem de máquina cujo desempenho é comparável ao de alguns dos mais conhecidos algoritmos da área foi apresentado. No entanto, a principal contribuição deste capítulo é a técnica utilizada na construção deste algoritmo; que consiste basicamente em acoplar a uma árvore-DND adaptativa, módulos para tratamento de valores contínuos, ausentes e inconsistentes. O próprio conceito de árvore-DND adaptativa aqui apresentado poderá ser futuramente utilizado na produção de diferentes estratégias para aprendizagem no domínio discreto (AdapTree é apenas um caso especial de árvore-DND adaptativa), utilizando, por exemplo, outros tipos de funções adaptativas.

O conceito de árvore-DND adaptativa oferece um nova forma de enxergar uma estratégia de aprendizagem: como algo (na caso uma árvore de decisão) capaz de se auto-modificar para se adaptar ao ambiente. Ou seja, o algoritmo de aprendizagem está agora embutido no modelo aprendido. Esse novo enfoque abre um interessante espaço para novas pesquisas em que o próprio algoritmo de aprendizagem, que agora faz parte do modelo aprendido, seja também objeto de modificações. O resultado disto, uma espécie de dispositivo adaptativo adaptativo, poderá vir a ser uma espécie de *meta-algoritmo* de aprendizagem, capaz da alterar dinâmicamente o seu funcionamento buscando desempenho máximo em diferentes domínios.

O módulo para tratamento de valores discretos do AdapTree-E, a AdapTree, é inerentemente incremental, uma vez que tanto o custo de treinamento, quanto de teste, dependem apenas do exemplo que está sendo lido (e não dos exemplos lidos anteriormente). No entanto, o desempenho da discretização e a substituição de valores ausentes são diretamente dependentes da quantidade de exemplos de treinamento disponíveis inicialmente. Se este conjunto de treinamento inicial for altamente representativo, os intervalos deduzidos na discretização, e o valores de moda e média calculados para substituição de valores ausentes, não precisarão ser mais reajustados e o AdapTree-E funcionará muito bem em modo incremental (com novos exemplos de treinamento podendo ser incorporados de forma intercalada com exemplos de teste). No entanto, novos estudos serão necessários para comparar o desempenho do AdapTree-E em relação à incrementabilidade, bem como para determinar estratégias ótimas que permitam ajustes temporários nos valores de intervalos de discretização, moda e média, com base nos novos exemplos de treinamento que vão sendo acumulados durante a execução incre-

mental. Estes novos estudos deverão incluir também a busca por técnicas eficientes para reordenação dos atributos, talvez adaptando algumas das técnicas apresentadas em (PICCHIA, 1993).

Duas outras linhas interessantes para futuras pesquisas incluem a busca por um mecanismo subjacente que possibilite um tratamento mais homogêneo para valores contínuos e discretos. Os autômatos híbridos (HENZINGER, 1996; HENZINGER; HO, 1993) parecem ser uma alternativa promissora, pois oferecem um formalismo que permite a modelagem tanto dos aspectos discretos, quanto dos aspectos contínuos, de um sistema híbrido (sistemas que tratam simultaneamente componentes digitais e analógicas). Outra linha a ser explorada está relacionada com a replicação e distribuição das ações adaptativas por diferentes nós de uma árvore de decisão. Esta característica poderá ser explorada futuramente em sistemas que permitam a execução distribuída e/ou paralela de funções adaptativas, visando à melhoria no desempenho global do dispositivo.

6 FUNÇÕES ADAPTATIVAS

Mostramos nesta seção como a execução de ações elementares de consulta pode ser tratada como um problema de satisfação seqüencial de restrições, SSR (ver cap. 4). Para isto, são propostas algumas alterações e refinamentos na definição original das funções adaptativas. O resultado será expresso através de um algoritmo que, além de servir como referência em estudos mais detalhados sobre o desempenho de dispositivos adaptativos, simplificará futuras implementações da camada adaptativa. De fato, uma implementação deste algoritmo já foi utilizada no desenvolvimento da ferramenta AdapTools (ver capítulo 8). Esta base teórica também poderá ser útil na análise da sensibilidade do desempenho e da expressividade de um dispositivo adaptativo, em relação a diferentes formas de restrições que podem ser impostas na especificação de funções adaptativas.

Uma outra opção natural para a formalização das funções adaptativas seria a álgebra relacional (CODD, 1970). No entanto, além de o cálculo de predicados ocupar um espaço bem mais destacado na área da inteligência artificial, foi demonstrado que a álgebra relacional é menos expressiva que o cálculo de predicados (MINKER, 1996). Além disto, os diversos trabalhos na área de *lógica e bancos de dados* (MINKER, 1996) permitem que conceitos de ambos os campos possam ser intercambiados, de forma que nossa opção pelo cálculo de predicados não impede o reaproveitamento dos resultados na álgebra relacional, principalmente em relação à otimização das operações de consulta.

6.1 Modelagem de Ações Adaptativas Elementares

O primeiro passo na modelagem de funções adaptativas usando SSR é a representação das regras do mecanismo subjacente como conjuntos de literais. Em um primeiro

momento, autômatos de estados finitos (AEF) serão assumidos como mecanismo subjacente. Uma regra em um AEF é uma transição que parte de um estado q_i e chega a um estado q_f, em resposta à leitura de um determinado símbolo a da cadeia de entrada (também não consideraremos neste ponto a existência de ações adaptativas associadas às transições). Para representar cada regra de um AEF em SSR utilizamos um predicado ternário chamado *regra*, cujos argumentos correspondem respectivamente ao estado de saída, símbolo de leitura e estado de chegada. Este é o único predicado necessário nesta modelagem. Para ilustrar, mostramos abaixo a base de conhecimentos em SSR que representa o AEF da figura 6.1 (a partir deste ponto adotaremos um pequena variação notacional para o cálculo de predicado: variáveis serão denotadas pelo prefixo "?", permitindo assim que nomes de constantes possam ser iniciados com letras minúsculas):

$$\{regra(q_0,a,q_0)\},\{regra(q_0,a,q_1)\},\{regra(q_0,a,q_2)\},\{regra(q_0,a,q_3)\},$$
$$\{regra(q_2,b,q_4)\},\{regra(q_4,c,q_2)\},\{regra(q_4,c,q_4)\},\{regra(q_4,c,q_5)\}$$

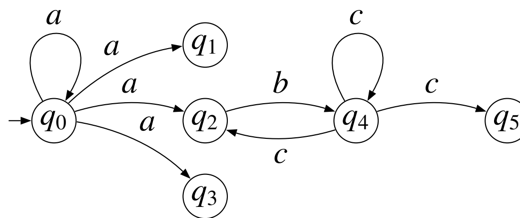


Figura 6.1: Exemplo de um autômato de estados finitos

É fácil perceber que qualquer base de conhecimento representando as transições de um autômato respeita as restrições da SSR. Antes de apresentar nossa proposta de representação para ações elementares iremos propor uma variante da definição de função adaptativa apresentada na seção 2.2.1, visando à obtenção de um modelo mais simples e que resolve algumas questões conceituais, relacionadas com o papel das variáveis em consultas elementares. A definição original [1], além de limitar em 1 o número de instanciações possíveis para cada variável, permite que operações essencialmente de consulta sejam executadas a partir de ações elementares de remoção -

[1]A análise aqui apresentada é baseada em uma possível interpretação para a definição de variáveis que consta da seção 4.2.1 de (NETO, 1993), em que se lê: "*...As variáveis...são preenchidas por ações adaptativas elementares de consulta ou de eliminação, e, uma vez preenchidas, mantêm permanentemente o valor assim recebido enquanto a função estiver sendo executada, não podendo portanto ser alteradas durante a execução da função...*", e não necessariamente na definição original. Utilizaremos o termo "definição original", por conveniência, quando nos referirmos a esta definição.

além das próprias ações elementares de consulta. Para exemplificar o problema com a definição original de variáveis mostramos abaixo um possível conjunto de ações elementares (usando triplas para representar os três elementos de uma transição em um AEF e o prefixo "?" para introduzir o nome de variáveis):

$$? \, [(?x, ?y, ?x)] \quad + [(?x, b, ?x)] \quad - [(q_0, a, ?z)]$$

O primeiro problema surge devido à restrição no número de instanciações de variáveis: quando aplicada ao autômato da figura 6.1, existem claramente duas transições que satisfazem a ação elementar de consulta, $?[(?x, ?y, ?x)]$, que são justamente os dois únicos laços do autômato. No entanto, apenas um destes laços pode ser utilizado para instanciar as variáveis. Por referir-se preferencialmente a dispositivos determinísticos (embora isto não esteja explicitado em (NETO, 1993)), a definição original comenta mas não deixa muito claro a maneira como este tipo de impasse deve ser resolvido. Este exemplo ilustra também o tratamento heterogêneo da operação de consulta, que acaba ocorrendo também durante a remoção, mas nesta, de maneira implícita, na busca pelo padrão $(q_0, a, ?z)$. Nossa proposta, além de restringir as operações de consulta às ações elementares de consulta, clareando assim o papel dos três tipos de ações elementares, retira a limitação em relação a instanciação de variáveis. O exemplo acima, deveria ser reescrito, seguindo nossa proposta, como:

$$? \, [(?x, ?y, ?x)] \;\; ? \, [(q_0, a, ?z)] \quad + [(?x, b, ?x)] \quad - [(q_0, a, ?z)]$$

A adoção de uma interpretação para funções adaptativas em que não existe limite no número de instanciações de uma variável nos permite propor um mecanismo de consulta mais natural, baseado em SSR e programação em lógica, em que as variáveis são instanciadas com todos os possíveis valores. Com isso, o conjunto de ações elementares de consulta de uma função adaptativa pode agora ser modelado como uma conjunção de literais "regra", e a execução de consultas, como provas de teorema em SSR, em que a base de conhecimento é o conjunto de transições do autômato, representadas como literais. No exemplo acima, as ações elementares de consulta equivalem, em SSR, à formula: $\{regra(?x, ?y, ?x)\}, \{regra(q_0, a, ?z)\}$, que é verdadeira em relação às seguintes oito substituições (utilizando a notação apresentada na seção 4.1 para cada substituição):

$\{?x/q_0, ?y/a, ?z/q_0\}$, $\{?x/q_0, ?y/a, ?z/q_1\}$, $\{?x/q_0, ?y/a, ?z/q_2\}$, $\{?x/q_0, ?y/a, ?z/q_3\}$,
$\{?x/q_4, ?y/c, ?z/q_0\}$, $\{?x/q_4, ?y/c, ?z/q_1\}$, $\{?x/q_4, ?y/c, ?z/q_2\}$, $\{?x/q_4, ?y/c, ?z/q_3\}$

O conjunto de substituições possíveis é completamente definido pela execução das consultas elementares. Na fase de execução das remoções, todas as ações elementares de remoção do conjunto são executadas independentemente, para cada substituição. A execução das inserções é efetuada de maneira similar à das remoções. A figura 6.2 mostra o resultado da execução das ações elementares, mostradas acima, sobre o autômato da figura 6.1.

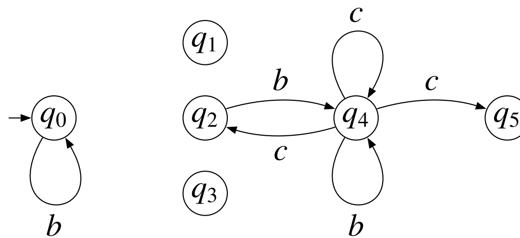


Figura 6.2: Autômato após a execução das ações elementares

A admissão de instanciação múltipla para variáveis abre uma nova possibilidade para a interpretação dos geradores. Pelas mesmas razões já mencionadas, e da mesma forma como as variáveis, na definição original os geradores carregam um único valor durante a execução de uma função adaptativa. Nesta proposta, com a instanciação múltipla de variáveis, também os geradores poderiam, a princípio, ser multiplamente instanciados. Dessa forma, símbolos diferentes seriam gerados para cada diferente substituição de variáveis. As figuras 6.3.(b) e 6.3.(c) mostram os autômatos resultantes da aplicação das ações adaptativas elementares abaixo (com geradores sendo identificados pelo prefixo "*") sobre o autômato da figura 6.3.(a), usando a definição original e a definição com múltiplas instâncias, respectivamente:

$$?\ [(q_1, ?x, ?y)] \quad +[(?y, ?x, *g)]$$

As duas definições são perfeitamente aplicáveis e, dependendo do tipo de resultado esperado pela execução da função adaptativa, poder-se-ia justificar facilmente a utilização de uma ou de outra. No entanto, a definição para geradores proposta neste trabalho (múltipla instanciação), além de ser mais coerente com a definição proposta para as variáveis (também multiplamente instanciadas), parece tornar mais poderoso o mecanismo de geração de símbolos. O maior poder da nova definição é

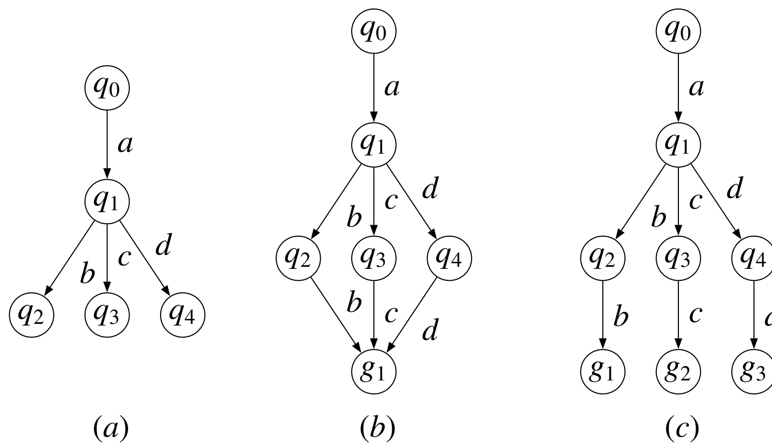


Figura 6.3: Diferentes interpretação para geradores
(a) Autômato Original (b) Geradores com Única Instância (c) Geradores com Múltiplas Instâncias

verificável constatando-se que o efeito de um gerador unicamente instanciado pode ser obtido facilmente através da utilização de constantes ou da passagem de parâmetros (os parâmetros continuam mantendo-se inalteráveis durante toda a execução da função adaptativa). Já o efeito inverso, de simular a criação de múltiplos novos símbolos utilizando um gerador unicamente instanciável, não pode ser trivialmente obtido.

Por exemplo, a função adaptativa abaixo, em que g_1 é um estado não referenciado por qualquer transição do autômato subjacente original, é uma nova versão para a função adaptativa definida acima. Com esta nova versão obtemos o efeito apresentado na figura 6.3.(b), mesmo utilizando a definição de geradores multiplamente instanciados (nesta nova versão simplesmente trocamos o gerador por uma constante). O efeito inverso (simular 6.3.(c) sem geradores multiplamente instanciados) já é bem mais complicado de ser obtido, pois a quantidade de novos estados a serem criados só é determinada em tempo de execução. Poderíamos, para o caso particular, utilizar 3 ações elementares de inserção - uma para cada diferente estado a ser criado (com um gerador diferente em cada ação) - no entanto, é fácil perceber que esta solução não pode ser generalizada.

$$?\ [(q_1, ?x, ?y)] \quad +\ [(?y, ?x, g_1)]$$

Ações adaptativas iniciais e finais também foram retiradas da definição, visando uma simplificação do modelo. Com isto, o efeito de seqüência que poderia ser obtido

através dessas ações passa a depender do mecanismo subjacente. Nos autômatos de estados finitos adaptativos, por exemplo, poderíamos usar seqüências adicionais de transições em vazio para carregar as ações adaptativas que, na formulação original, permaneceriam incluídas na definição de uma função adaptativa específica. Justifica-se este tipo de simplificação pelo fato de que, em geral, o usuário da tecnologia adaptativa já domina a utilização do mecanismo subjacente. Para principiantes em dispositivos adaptativos, uma camada adaptativa mais simples pode significar um menor esforço de aprendizagem.

Seguindo uma prática comum na programação em lógica, optamos pela declaração implícita de variáveis, cujos nomes devem sempre ser precedidos pelo símbolo de interrogação. É importante notar que a semântica das ocorrências de variáveis nas ações elementares de consulta, nesta proposta, é diferente daquelas que ocorrem em ações elementares de inserção e remoção. No primeiro caso, as variáveis são associadas a valores obtidos pela inspeção do autômato subjacente, enquanto na caso das inserções e remoções, os valores associados são apenas utilizados e nunca alterados. A restrição das operações de consulta às ações elementares de consulta retira qualquer possibilidade de confusão na interpretação da semântica das variáveis (o que não acontece na definição original). A ocorrência de uma variável em uma ação elementar de inserção ou remoção que não tenha ocorrido previamente em uma ação elementar de consulta resultará na inibição da execução da inserção ou remoção correspondente.

Por último, propomos que os parâmetros de uma função adaptativa sejam também implicitamente declarados, e acessados, quando necessário, através dos símbolos reservados $\%_1, \%_2, ..., \%_k$, que correspondem, seqüencialmente, aos k valores informados como argumentos na chamada da função adaptativa. A ocorrência de parâmetros indefinidos em uma ação elementar também resultará na inibição da execução de tal ação.

O algoritmo 6.1 mostra como uma função adaptativa pode ser executada, de acordo com a proposta apresentada neste trabalho. A execução das ações elementares de consulta, baseada em uma versão simplificada de unificação e em resolução ordenada, é descrita separadamente no algoritmo 6.2. A versão simplificada de unificação é possível devido à inexistência de expressões funcionais entre os termos do único predicado utilizado para representar transições e ações elementares de consulta. Além disto, um dos predicados a ser unificado, representando as transições, é sempre livre

Algoritmo 6.1 Executa função adaptativa

entrada: Transições do autômato original $T = t_1, ..., t_n$
Ações elementares de consulta $A_C = c_1, ..., c_{n_c}$
Ações elementares de remoção $A_R = r_1, ..., r_{n_r}$
Ações elementares de inserção $A_I = i_1, ..., i_{n_i}$
Lista de parâmetros P
saída: Lista T' com as transições do autômato após modificações

1: $T' \leftarrow T$
2: Substitui parâmetros em A_C, A_R e A_I usando P
3: **se** $n_c > 0$ **então**
4: $S \leftarrow$ ExecutaConsultas(T,A_C,Ø) {S recebe lista de substituições}
5: **se** $S \neq \emptyset$ **então**
6: **para** $j = 1$ até n_r **faça**
7: **para todo** $s \in S$ **faça**
8: Aplica substituição s sobre r_j
9: $T' \leftarrow T' - r_j$
10: **fim para**
11: **fim para**
12: **para** $j = 1$ até n_i **faça**
13: **para todo** $s \in S$ **faça**
14: Aplica substituição s sobre i_j
15: Gera um novo símbolo para cada diferente gerador de i_j e efetua substituições correspondentes
16: $T' \leftarrow T' + i_j$
17: **fim para**
18: **fim para**
19: **fim se**
20: **senão** $\{n_c = 0\}$
21: Remover de A_R e A_I todas as ações contendo variáveis
22: Executar inserções e remoções, agora sem variáveis, listadas em A_R e A_I
23: **fim se**

de variáveis, o que torna trivial o cálculo do unificador mais geral.

Algoritmo 6.2 ExecutaConsultas(T, A_C, s)

entrada: Transições do autômato original $T = t_1, ..., t_n$
Ações elementares de consulta $A_C = c_1, ..., c_{n_c}$
Lista s com variáveis instanciadas até o momento
saída: Lista S de substituições

1: Seja S a lista de substituições que será retornada pelo algoritmo { Para evitar a utilização de mais um parâmetro, S deve possuir escopo global e ser inicializada com vazio, antes do início do algoritmo }
2: **se** $n_c = 0$ **então** {Foi encontrada uma substituição s que satisfaz a consulta}
3: $S \leftarrow S \cup \{s\}$
4: **senão**
5: **para** $i = 1$ até n **faça**
6: **se** $s' \Leftarrow$ Unifica(c_1, t_i, s) **então**
7: ExecutaConsultas($T, A_C - c_1, s'$)
8: **fim se**
9: **fim para**
10: **fim se**

6.2 Funcionamento do Algoritmo de Execução de Função Adaptativa

Ilustraremos o funcionamento do algoritmo de execução de função adaptativa utilizando o autômato adaptativo mostrado na figura 6.4. O mecanismo subjacente (figura 6.4.(a)) é representado usando a notação padrão para autômatos de estados finitos, com exceção do rótulo $[.F(a, b)]$, ligado a única transição que parte do estado inicial do autômato. Este rótulo indica que a função adaptativa F, com parâmetros a e b, deve ser executada logo após a transição para o estado q_1, tendo para isso consumido o símbolo a. A representação de função adaptativa F (figura 6.4.(b)) é inspirada na notação utilizada em redes semânticas, com variáveis, geradores e parâmetros implícitos, denotados conforme sugerido anteriormente (prefixos "?","*" e "%", respectivamente). As duas setas paralelas têm como origem os padrões a serem procurados no autômato subjacente e como destino o resultado das modificações a serem aplicadas sobre os padrões encontrados. A função adaptativa F pode ser descrita também pela seguinte lista de ações adaptativas elementares:

$$?\,[(?x, \%_1, ?y)] \quad ?\,[(?y, \%_2, ?z)] \quad -\,[(?x, \%_1, ?y)]$$

Algoritmo 6.3 Unifica(c, t, s)

entrada: Lista c com os termos da consulta
Lista t com os termos da transição
Lista s com as variáveis instanciadas (*binded*) até o momento
saída: Valor verdade indicando se c e t são unificáveis
Lista s' com variáveis instanciadas após unificação

1: Seja $c = c_1, c_2, ..., c_m$ e $t = t_1, t_2, ..., t_m$
2: $s' \leftarrow s$
3: $Retorno \leftarrow VERDADEIRO$
4: **para** $j = 1$ até m **faça**
5: **se** c_j é uma variável **então**
6: **se** c_j aparece instanciada em s' **então**
7: $c_j \leftarrow$ valor de c_j em s' {c_j deixa de ser uma variável}
8: **senão**
9: $s' \leftarrow s \cup \{c_j/t_j\}$
10: $c_j \leftarrow t_j$
11: **fim se**
12: **fim se**
13: **se** $c_j \neq t_j$ **então**
14: $Retorno \leftarrow FALSO$
15: **fim se**
16: **fim para**

$$-[(?y, \%_2, ?z)] \quad +[(?x, \%_1, ?z)] \quad +[(?x, \%_2, ?x)]$$

A entrada do algoritmo 6.1, quando aplicada pela primeira vez ao autômato do exemplo, é a seguinte (omitimos o nome do predicado, *regra*, mostrando apenas seus argumentos):

$$\begin{aligned}
T &= \{(q_0, a, q_1), (q_1, a, q_2), (q_2, a, q_3), (q_3, b, q_4), \\
&\quad (q_4, a, q_5), (q_5, a, q_6), (q_6, a, q_7), (q_7, b, q_0)\} \\
A_C &= \{(?x, \%_1, ?y), (?y, \%_2, ?z)\} \\
A_R &= \{(?x, \%_1, ?y), (?y, \%_2, ?z)\} \\
A_I &= \{(?x, \%_1, ?z), (?x, \%_2, ?x)\} \\
P &= \{a, b\}
\end{aligned}$$

Inicialmente, durante a execução do algoritmo 6.1, temos a substituição de parâ-

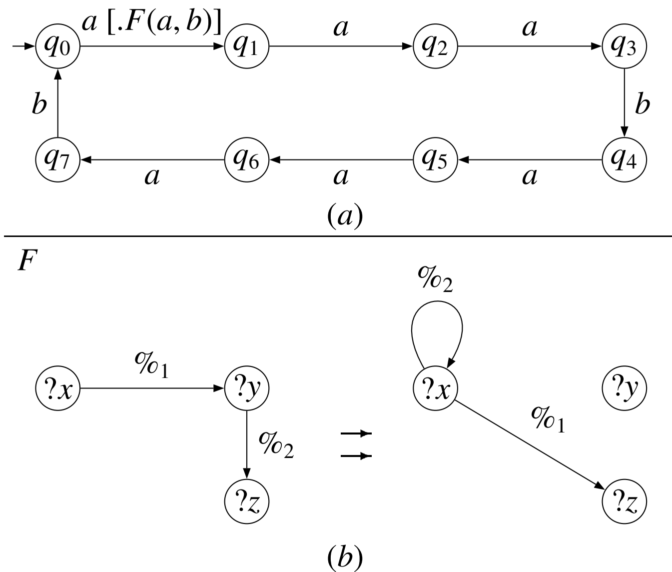


Figura 6.4: Autômato adaptativo antes da execução do algoritmo 6.1
(a) Autômato Subjacente (b) Camada Adaptativa

metros (linha 2), que instancia as listas de ações elementares para:

$$\begin{aligned} A_C &= \{(?x, a, ?y), (?y, b, ?z)\} \\ A_R &= \{(?x, a, ?y), (?y, b, ?z)\} \\ A_I &= \{(?x, a, ?z), (?x, b, ?x)\} \end{aligned}$$

Como existem ações elementares de consulta ($n_c = 2$), o algoritmo 6.2 é acionado (linha 4). A figura 6.5 mostra a árvore completa de chamadas recursivas do algoritmo 6.2, junto com os parâmetros passados em cada chamada. As folhas marcadas com contorno em linhas duplas indicam que a recursão foi interrompida pela determinação de uma substituição válida (linhas 2 e 3 do algoritmo 6.2). As outras folhas correspondem a falhas (retorno de um valor FALSO) no teste da linha 6 (algoritmo 6.2), ou seja, consulta e transição não puderam ser unificadas.

Retornando ao algoritmo 6.1, como o conjunto de substituições S contém dois elementos, $\{?x/q_2, ?y/q_3, ?z/q_4\}$ e $\{?x/q_6, ?y/q_7, ?z/q_0\}$, temos a execução dos laços de remoção e inserção de transições. A aplicação das substituições (linhas 8 e 12) é bem simples, e consiste apenas em percorrer toda a ação elementar, substituindo cada variável pelo valor correspondente, apontado em s (a ação elementar deve retornar ao seu estado original após a execução da operação correspondente, para que cada substituição ocorra de forma independente). A linha 13 não tem qualquer efeito neste

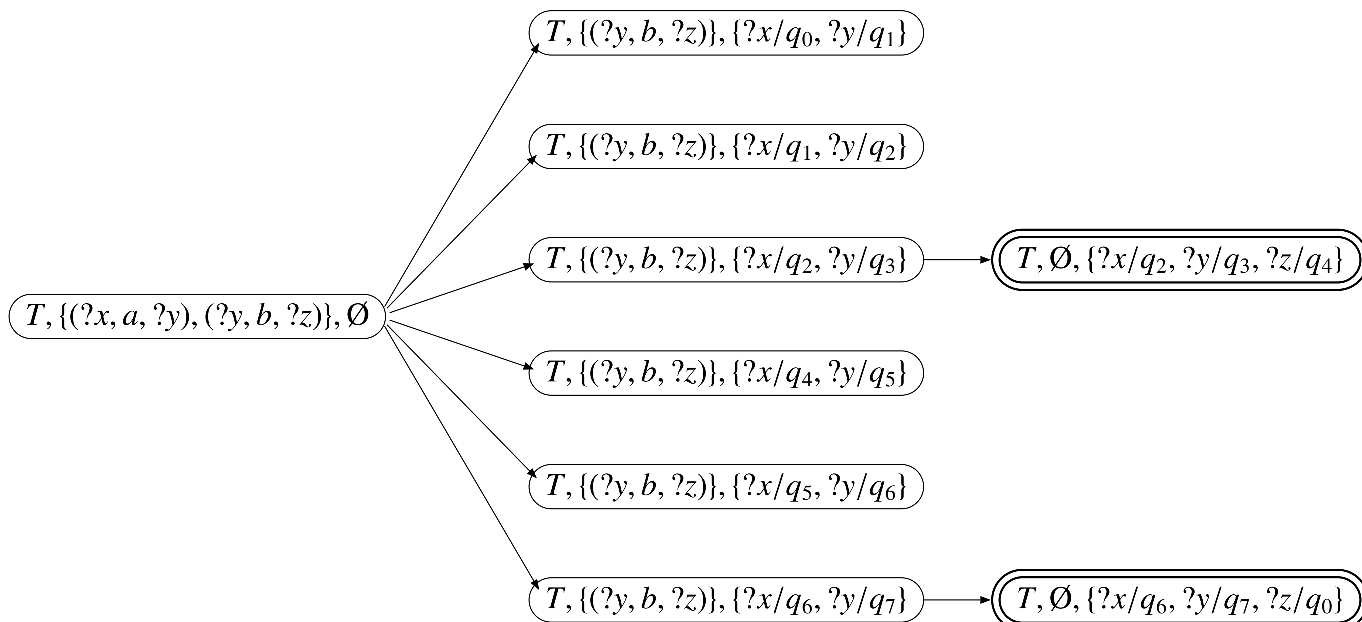


Figura 6.5: Árvore de chamadas recursivas do algoritmo 6.2

exemplo, pois as ações elementares de inserção não apresentam geradores. O autômato resultante é mostrado na figura 6.6.

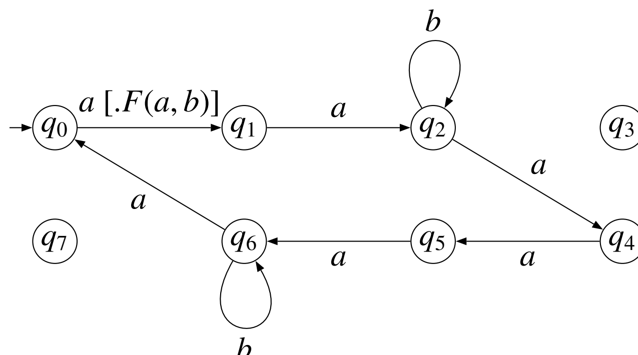


Figura 6.6: Autômato subjacente após a execução do algoritmo 6.1

6.3 Considerações sobre a ordem de aplicação das ações elementares de consulta

O algoritmo 6.1 nada mais é que um caso particular da técnica de prova automática de teorema, denominada resolução ordenada, e portanto, assim como na resolução ordenada, a ordem dos literais que compõem o teorema não afeta o resultado obtido (GENESERETH; NILSSON, 1988). Em outros termos, a ordem das ações elementares de

consulta não tem qualquer influência sobre o conjunto das transições que são inseridas ou removidas do autômato original. No entanto, a ordem das consultas pode alterar significativamente o desempenho do algoritmo, como se pode perceber pela árvore de chamadas recursivas (figura 6.7) obtida para o mesmo exemplo da seção anterior, tendo sido invertida a ordem das consultas fazendo $A_C = \{(?y, \%_2, ?z), (?x, \%_1, ?y)\}$. A escolha da melhor ordem a ser adotada é um problema complexo, e depende muitas vezes de conhecimento que se tem acerca do problema que está sendo representado através do autômato. Um estudo aprofundado sobre o problema da ordenação de literais em prova de teoremas pode ser encontrado em (LEDENIOV; MARKOVITCH, 1998) e (SMITH, 1985).

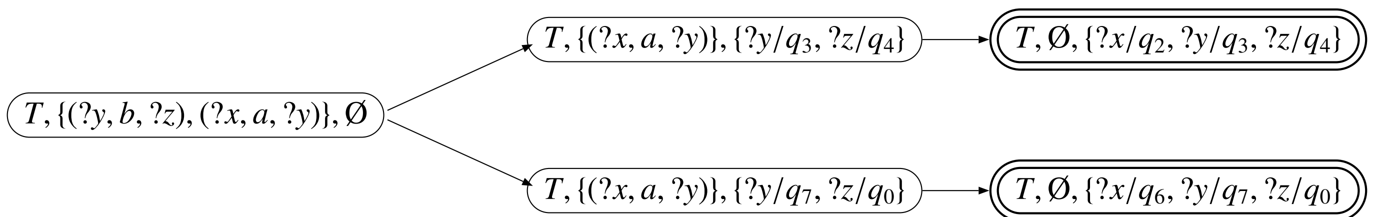


Figura 6.7: Árvore de chamadas recursivas com outra ordenação para consultas

6.4 Complexidade do Algoritmo de Execução de Função Adaptativa

Calcularemos informalmente nesta seção o custo, no pior caso, do algoritmo 6.1 em relação aos tamanhos das listas de transição (n), ações elementares de consultas (n_c), ações elementares de remoção (n_r), ações elementares de inserção (n_i) e parâmetros (n_p). A linha 1, que executa uma simples cópia da lista de transições T, de tamanho n, pode trivialmente ser executado em $O(n)$ passos. A linha 2 pode ser executada percorrendo-se as três listas de ações elementares, cujo tamanho somado é exatamente $n_c + n_r + n_i$, e efetuando-se uma busca na lista de parâmetros a cada passo do percurso (para se determinar o valor do parâmetro). Como a lista de parâmetros possui tamanho n_p, temos que o custo de execução da linha 2 é da ordem de $O((n_c + n_r + n_i)n_p)$. Um resultado positivo para o teste da linha 3 é certamente o pior caso, visto que a execução das linhas 16 e 17 é um caso especial do que ocorre entre as linhas 4 e 14.

Na linha 4 temos a chamada do algoritmo recursivo *ExecutaConsulta*, cujo custo é determinado basicamente pelo teste da unificação (linha 6). É fácil perceber que

o maior número de unificações é obtido quando todos os elementos da consulta são variáveis e cada variável ocorre uma única vez em todo o conjunto de consultas, ou seja, quando $A_c = \{(?x_1, ?x_2, ?x_3), (?x_4, ?x_5, ?x_6), ..., (?x_{k-2}, ?x_{k-1}, ?x_k)\}$ e $x_i \neq x_j$ para todo $1 \leq i, j \leq k$ com $i \neq j$. Neste caso, temos uma chamada recursiva para cada um dos n passos do laço da linha 5. A execução do algoritmo de unificação não depende do tamanho da entrada do algoritmo 6.1 e seu custo pode ser considerado constante (na verdade, depende apenas da quantidade de elementos da uma transição - neste caso - 3). Como a cada recursão temos a eliminação de um dos elementos do conjunto A_C, de ações elementares de consulta, e o teste da linha 2 interrompe uma seqüência de recursões quando A_C é vazio, temos que, no pior caso, o custo do algoritmo 6.2 é da ordem de $O(n^{n_c})$ (n execuções da linha 7 no primeiro nível da recursão, multiplicado por n execuções da linha 7 no segundo nível da recursão, e assim por adiante, até o nível máximo de recursão que é igual ao número de ações elementares de consulta, n_c). O tamanho do conjunto de substituições S, que será retornado pelo algoritmo 6.2, também é da ordem de $O(n^{n_c})$, visto que no pior caso, todas as seqüências de recursões terminam na linha 3 (e portanto geram, cada uma, a inserção de um elemento em S). Uma outra maneira de se chegar a este resultado é percebendo que o tamanho de S, no pior caso, é igual ao tamanho do produto cartesiano n_c-ário T^{n_c}. O tamanho deste produto cartesiano é obviamente n^{n_c}, uma vez que $|T| = n$.

Retornando ao algoritmo 6.1, percebe-se que a execução das ações elementares de remoção envolve um aninhamento de laços (linhas 6 e 7). Assumindo que o custo da remoção de cada transição esteja na ordem de $O(n)$ (remoção de um elemento de uma lista), temos que a execução das remoções custa $O(n_r n^{n_c} n)$ (total de ações elementares de remoção, n_r, multiplicado pelo tamanho do conjunto de substituições, $O(n^{n_c})$, multiplicado pelo custo de cada remoção, $O(n)$). Como veremos abaixo, embora o custo das remoções tenha sido super-estimado (estamos admitindo, por exemplo, um custo maior que o total de transições do autômato), uma redução neste custo não afetaria a ordem de grandeza do custo global do algoritmo. O custo da execução das ações elementares de inserção é muito parecido, e está na ordem de $O(n_i n^{n_c} n)$. Como os custos obtidos ocorrem em seqüência, temos um custo total para o algoritmo 6.1 igual a:

$$O(n) + O((n_c + n_r + n_i)n_p) + O(n^{n_c}) + O(n_r n^{n_c} n) + O(n_i n^{n_c} n)$$

Fazendo $k = max(n_c, n_r, n_i, n_p)$, retirando termos claramente limitados superior-

mente por outro termos, e assumindo $k \geq 1$, podemos simplificar a fórmula acima para:

$$O(n) + O(3k^2) + O(n^k) + 2 * O(kn^{k+1}) = O(3k^2 + kn^{k+1})$$

Em relação ao comportamento assintótico da função acima, é razoável considerar que os valores n_c, n_r, n_i e n_p são tão pequenos em relação a n que podem ser desprezados. Aliás, se o interesse principal se concentra no desempenho do dispositivo adaptativo, como um todo, o único valor variável é justamente n (a menos que estejamos considerando um dispositivo adaptativo cujo mecanismo adaptativo seja também, por sua vez, outro dispositivo adaptativo. Neste caso, o conjunto de ações elementares pode também ter seu tamanho aumentando em função do tamanho da entrada). Considerando n_c, n_r, n_i e n_p constantes, a fórmula acima pode ser reduzida a $O(n^{k+1})$, ou seja, um custo polinomial de ordem $k+1$. É importante ressaltar que mesmo utilizando estruturas de dados mais eficientes, que permitam operações de busca e inserção em tempos logarítmicos e até constantes, o custo do pior caso do algoritmo 6.1 poderá ser melhorado, no máximo, para $O(n^k)$. Isto acontece porque o custo do algoritmo é dominado assintoticamente pelo tamanho do conjunto S, retornado pelo algoritmo 6.2, que continuará sendo, no pior caso, da ordem de $O(n^k)$. Neste caso, o k corresponderá exatamente ao total de ações elementares de consulta.

6.5 Considerações sobre Cálculo de Complexidade de Dispositivos Adaptativos

Demonstramos na seção anterior que o custo da execução de uma função adaptativa, no pior caso, depende polinomialmente do tamanho do conjunto de transições do autômato subjacente. No entanto, ao calcular a complexidade de um autômato (seja ele adaptativo ou não), a variável que nos interessa é o tamanho da cadeia de entrada. Em autômatos convencionais, com um conjunto estático de transições, o custo se restringe à contagem das transições realizadas em função do tamanho da entrada. Em autômatos adaptativos, no entanto, o tamanho do conjunto de transições varia indiretamente em função do tamanho da entrada, e conseqüentemente, o custo de execução das funções adaptativas não pode ser considerado sempre constante.

Mostraremos agora um exemplo extremo de como a execução de funções adaptativas pode afetar severamente o desempenho de um autômato adaptativo. Na figura 6.8 temos um autômato adaptativo bastante simples que aceita a linguagem a^* sobre o alfabeto $\{a, b\}$. Se considerássemos constante o custo de execução da função adaptativa F, a complexidade deste autômato, no pior caso (quando a entrada consiste de uma seqüência de símbolos a), seria da ordem de $O(n)$ (linear), em que n é o tamanho da cadeia de entrada.

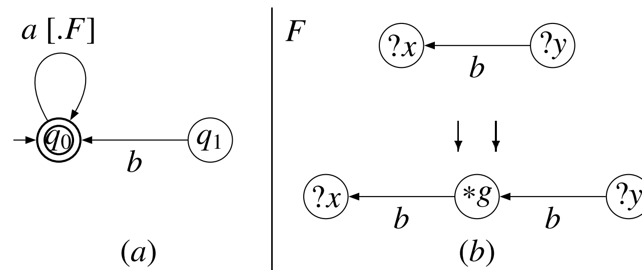


Figura 6.8: Autômato adaptativo com complexidade exponencial no tempo (a) Autômato Subjacente (b) Camada Adaptativa

No entanto, a função adaptativa F, que troca cada transição que consome um símbolo b, por duas novas transições consumindo b, faz com que o tamanho do conjunto de transições dobre a cada leitura de um símbolo a. Assumindo que m seja o tamanho do conjunto de transições, temos que, no pior caso, $m = 1 + 2^n$ (a constante 1 corresponde a transição que consome o símbolo a). Aplicando a fórmula simplificada para cálculo do custo de complexidade obtida na seção anterior, com $k = 1$ (apenas uma ação elementar de consulta), temos que a execução de cada função adaptativa é da ordem de $O(m^k) = O(m^1) = O(m) = O(1 + 2^n) = O(2^n)$. Como esta função adaptativa é executa para cada a consumido, temos um custo total para o autômato adaptativo na ordem de $O(n2^n)$, ou seja, um custo exponencial.

No cálculo de complexidade mostrado acima, é interessante notar que o passo fundamental foi encontrar uma função que relaciona o tamanho da cadeia de entrada com o tamanho do conjunto de transições. A mesma estratégia poderá ser utilizado no cálculo de complexidade de outros autômatos adaptativos, no entanto, especial cuidado deverá ser tomado ao se analisar autômatos mais complexos. Funções adaptativas diferentes podem estar atuando sobre o mesmo mecanismo subjacente, complicando ainda mais a obtenção de uma fórmula que relacione os tamanhos da cadeia de entrada e do conjunto de transições.

O desempenho exponencial para certos problemas é um resultado esperado, uma vez que os autômatos adaptativos possuem poder de expressão de máquinas de Turing. Porém, o algoritmo apresentado neste trabalho é uma importante ferramenta na busca por classes de problemas que podem ser resolvidos de maneira eficiente. Uma classe interessante é formada pelos problemas que podem ser resolvidos por autômatos adaptativos cujas funções adaptativas não utilizam variáveis. Exemplos deste tipo de autômato, juntamente com indicações de sua importância em problemas relacionados com construção de compiladores, podem ser encontrados em (NETO, 1993). Analisaremos abaixo o desempenho desta classe de autômatos.

Em primeiro lugar, é preciso notar que autômatos sem variáveis não possuem ações elementares de consulta (que só têm sentido quando possuem variáveis). Portanto, a análise da complexidade do algoritmo de execução de funções adaptativas se restringe às linhas 1, 2, 16 e 17. As linhas 2 e 16 realizam operações que certamente não dependem do tamanho da cadeia de entrada: são operações sobre as listas de ações elementares, que possuem tamanho constante. Já as linhas 1 e 17 realizam operações sobre a lista de transições, que como vimos no exemplo acima, podem crescer até exponencialmente em função do tamanho da entrada. Veremos abaixo, no entanto, que para o caso especial em questão, este crescimento é linear em função da entrada.

Como não existem ações elementares de consulta, podemos concluir que cada ação elementar de inserção gerará uma única operação de inserção na lista de transições (não existem multiplas instanciações). Portanto, como o número de ações elementares de inserção é constante, o aumento no tamanho do conjunto de transições a cada execução de uma função adaptativa também é constante. Finalmente, como cada transição do autômato subjacente executa no máximo duas funções adaptativas, temos que, no pior caso, o tamanho de conjunto de transições cresce de maneira diretamente proporcional ao tamanho da cadeia de entrada (assumindo que o mecanismo subjancente seja sempre determinístico). Resumindo, se m é o tamanho do conjunto de transições e n é o tamanho da entrada, temos que, $m = O(n)$.

A execução da linha 1 apresenta, no pior caso, complexidade igual à O(n), pois realiza apenas uma cópia da lista de transições de tamanho $O(n)$. Se considerarmos que o mecanismo subjacente se mantém sempre deterministico, e portanto, com complexidade linear no tempo, $O(n)$. Temos que o impacto da execução da linha 1, no autômato adaptativo como um todo, é, no pior caso, da ordem de $O(n^2)$. Como estamos reali-

zando uma análise de complexidade no pior caso, é razoável supor que as operações de inserção e remoção da linha 17 não poderão ser executadas em um tempo melhor que logarítmico em função do tamanho do conjunto de transições (utilizando-se estruturas de dados apropriadas para realização de buscas binárias [2]). Portanto, o impacto da linha 17 na complexidade global do dispositivo adaptativo é da ordem de $O(nlog(n))$. Temos então uma complexidade resultante na ordem de $O(n^2 + nlog(n)) = O(n^2)$. Se retirarmos a linha 1 do algoritmo e passarmos a trabalhar diretamente no conjunto de transições original, conseguiremos reduzir este custo para $O(nlog(n))$. Com isto, conseguimos demonstrar que existem classes de autômatos adaptativos interessantes que podem ser executados de maneira eficiente. Estudos mais detalhados buscando novas classes de autômatos adaptativos e problemas que podem ser resolvidos através destas classes de autômatos deverão ser realizados no futuro. A contribuição maior da formalização apresentada neste capítulo é justamente a disponibilização de novos subsídios para um melhor direcionamento destes estudos.

6.6 Conclusão

Apresentamos neste capítulo uma nova proposta para formalização de funções adaptativas em dispositivos adaptativos, que tem como base a satisfação seqüencial de restrições e o algoritmo de unificação. Esta nova formalização nos permitiu um aprofundamento na análise de desempenho de funções adaptativas, bem como a introdução de simplificações notacionais e conceituais, como a utilização de variáveis, geradores e parâmetros implicitamente declarados, e uma melhor distinção conceitual entre as ações elementares de consulta e remoção. Também analisamos novas alternativas de interpretação da proposta original para autômatos adaptativos, principalmente em relação à forma em que variáveis e geradores devem ser instanciados em tempo de execução. Concluímos que a utilização de instanciação múltipla, além de ser mais natural, oferece ao usuário uma maior maleabilidade na especificação do dispositivo.

Como a SSR é uma especialização do cálculo de predicados, esta nova proposta oferece uma base sobre a qual características mais sofisticadas do cálculo de predicados, como negações e disjunções, poderão ser exploradas mais facilmente no futuro. Uma nova técnica para análise de desempenho de autômatos adaptativos foi ilustrada

[2]Como estamos realizando um cálculo de complexidade no pior caso, mesmos a utilização de tabelas de dispersão (*hash tables*) não garantiriam um custo melhor que logarítmico

através de exemplos. Esta técnica nos permitiu vizualizar casos particulares de dispositivos adaptativos cujo desempenho é seguramente polinominal no tempo. O presente trabalho representa um ponto de partida para a busca de novas formas de classificação de dispositivos adaptativos, relacionando expressividade e desempenho, com diferentes tipos de restrições a serem impostas ou retiradas da definição das ações elementares e dos algoritmos apresentados neste capítulo. A reformulação das árvores de decisão adaptativas, apresentadas na seção anterior, utilizando a proposta introduzida neste capítulo é também um importante objetivo a ser atingido no futuro.

7 AUTÔMATOS DE ESTADOS FINITOS ADAPTATIVOS

Apresentaremos neste capítulo um dispositivo adaptativo em que o mecanismo subjacente é um autômato de estados finitos (AEF). A escolha do autômato de pilha estruturado (APE) como mecanismo subjacente do primeiro e mais conhecido dispositivo adaptativo, o autômato adaptativo, parece ter tido um relação direta com os tipos de problema tratados nos primeiros trabalhos, na sua maioria relacionados com a construção de compiladores. A progressão de autômatos de estados finitos (linguagens regulares), para autômatos de pilha estruturados (linguagens livres-de-contexto) e para autômatos adaptativos (linguagens irrestritas), através do acréscimo de recursos que podem facilmente ser ignorados em problemas em que os mesmos não são necessários, é realmente uma solução atraente, principalmente para problemas diretamente relacionados com o processamento de linguagens. No entanto, como os autômatos de pilha estruturados não são tão difundidos quanto os autômatos de estados finitos, a apresentação e assimilação do conceito de dispositivo adaptativo acaba sendo dificultada pela necessidade de se apresentar também o mecanismo subjacente.

Com a crescente utilização de dispositivos adaptativos na solução de problemas que não são diretamente relacionados com linguagens, acreditamos ser interessante a formalização de um dispositivo adaptativo cujo mecanismo subjacente seja mais simples, e, de preferência, bastante conhecido, ao menos na área da computação. O autômato de estados finitos parece preencher muito bem estes requisitos. Mostraremos na próxima seção uma proposta de formalização para autômatos de estados finitos adaptativos que busca seguir o formato utilizado freqüentemente em livros-texto na área de teoria da computação. Serão mostrados também alguns exemplos simples de autômatos de estados finitos adaptativos.

7.1 Formalização

Além de utilizar um mecanismo subjacente mais simples que aquele usado nos autômatos adaptativos, esta proposta visa também simplificar a definição da camada adaptativa, retirando a especificação explícita de ações elementares de consulta e definindo as ações elementares de inserção e remoção através de funções e relações. Formalmente, definimos um autômato de estados finitos adaptativo ($\mathscr{A}$-AEF), como uma 8-upla M = $\langle Q, \Sigma, q_0, F, \delta, Q_\infty, \Gamma, \Pi \rangle$ em que os 5 primeiros elementos determinam o mecanismo subjacente, um AEF:

$\boldsymbol{Q \subseteq Q_\infty}$ é um conjunto finito e não-vazio de estados.

$\boldsymbol{\Sigma}$ é o alfabeto de entrada, também finito e não-vazio

$\boldsymbol{q_0 \in Q}$ é o estado inicial do autômato.

$\boldsymbol{F \subseteq Q}$ é o conjunto de estados finais.

$\boldsymbol{\delta \subseteq (Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty)}$ é a relação de transição, que inclui também transições em vazio (ϵ)

E os 3 elementos restantes representam a componente adaptativa do dispositivo:

$\boldsymbol{Q_\infty}$ conjunto contavelmente infinito de estados (este conjunto inclui estados que não são referenciados na configuração inicial do autômato mas que poderão ser eventualmente "incluídos" posteriormente).

$\boldsymbol{\Gamma \subseteq (\{+, -\} \times (Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty))}$ é o conjunto de todas as possíveis ações adaptativas elementares de inserção (+) e remoção (-).

$\boldsymbol{\Pi : (Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty) \rightarrow 2^\Gamma}$ é uma função que mapeia cada transição de δ em um conjunto de ações adaptativas elementares. Transições "normais" do autômato subjacente (que não geram a execução de ações adaptativas elementares) devem ser mapeadas para o conjunto vazio.

Para cada transição $x \in \delta$ em que $\Pi(x) = \emptyset$, o $\mathscr{A}$-AEF se reduz funcionalmente a um simples AEF (com eventuais transições em vazio). Nos outros casos, a transição

deve ser precedida da execução das ações adaptativas elementares definidas por Π. Visando simplificar ainda mais o dispositivo, para fins pedagógicos, retiramos do modelo as ações adaptativas posteriores. A utilização exclusiva de ações adaptativas anteriores não prejudica a capacidade expressiva do modelo resultante, pois existem artifícios, usando transições vazias, que podem ser empregados na simulação de ações adaptativas posteriores (IWAI, 2000). Definimos uma configuração de um $\mathscr{A}$-AEF como sendo uma quádrupla (q, w, K, Δ) pertencente à relação $Q_\infty \times \Sigma^* \times 2^{Q_\infty} \times 2^{(Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty)}$, na qual:

$\boldsymbol{q \in Q_\infty}$ é o estado corrente,

$\boldsymbol{w \in \Sigma^*}$ é a parte da cadeia de entrada ainda não lida,

$\boldsymbol{K \in 2^{Q_\infty}}$ é o conjunto de estados explicitamente referenciados pelo conjunto corrente de transições e

$\boldsymbol{\Delta \in 2^{(Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty)}}$ é o conjunto corrente de transições.

A relação de passo do autômato, $\vdash_M$, é definida pelo conjunto de pares ordenados (C_1, C_2) em que $C_1 = (q, w, K, \Delta)$ e $C_2 = (q', w', K', \Delta')$. Além disto, dizemos que $C_1 \vdash_M C_2$, se e somente se:

- $w = aw'$ para algum $a \in \Sigma \cup \{\epsilon\}$,
- $\delta(q, a) = q'$ e
- As ações adaptativas elementares do conjunto $\Pi((q, a, q'))$ modificam o conjunto de estados de K para K', e o conjunto de transições de Δ para Δ'. A ordem de aplicação das ações adaptativas elementares, que incluem apenas ações de inserção e remoção de transições, não afetam o resultado [1] .

Uma cadeia $w \in \Sigma^*$ é *aceita* por M se, e somente se, existe um estado $q \in F$ tal que $(q_0, w, Q, \delta) \vdash_M^* (q, \epsilon, K'', \Delta'')$, com K'' e Δ'' sendo qualquer conjunto de estados e transições ($\vdash_M^*$ denota, como é usual, o fechamento transitivo reflexivo da relação de passo do autômato). Nas próximas seções mostraremos alguns exemplos de autômatos de estados finitos adaptativos:

[1]Remoções de transições inexistentes ou incluídas por ações elementares de inserção durante o mesmo passo da execução da remoção devem ser ignoradas

7.2 Um $\mathscr{A}$-AEF que reconhece uma linguagem que não é livre de contexto

A figura 7.1.(a) mostra um $\mathscr{A}$-AEF M $= \langle Q, \Sigma, q_0, F, \delta, Q_\infty, \Gamma, \Pi \rangle$ que reconhece a clássica linguagem dependente de contexto, $a^n b^n c^n$. O símbolo Π superscrito em algumas das transições do autômato da figura 7.1 indica que a transição está ligada a uma seqüência não-vazia de ações adaptativas elementares, no caso:

$$\Pi((q_0, a, q_0)) = \{(-, p, \epsilon, q), (+, p, b, p'), (+, p', \epsilon, p''), (+, p'', c, q)\}, \forall p, q \in Q, p', p'' \notin Q \tag{7.1}$$

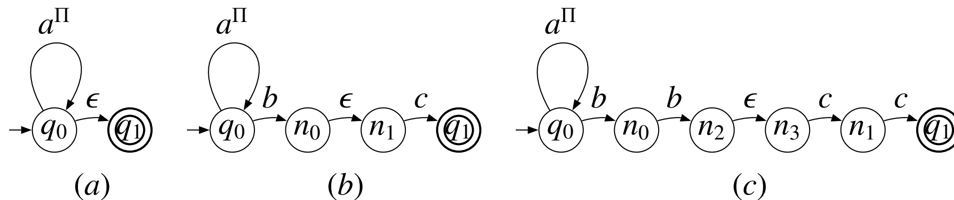


Figura 7.1: Autômato de estados finitos adaptativo que reconhece $a^n b^n c^n$
(a) Configuração inicial (b) Após primeira adaptação (c) Após segunda adaptação

A operação deste autômato ocorre da seguinte maneira: para cada símbolo *a* lido, o autômato substitui a única transição vazia do autômato por uma seqüência de 3 transições, a primeira consumindo o símbolo *b*, a segunda vazia (garantindo assim a existência de uma transição vazia), e a terceira consumindo o símbolo *c*. Ou seja, se *i* símbolos *a* forem lidos, teremos um caminho único ligando o estado inicial ao estado final do autômato, que reconhece exatamente a cadeia $b^i c^i$. As figuras 7.1.(b) e 7.1.(c) mostra as mudanças na estrutura do autômato que ocorrem após a leitura de dois símbolos *a* consecutivos.

7.3 Aprendizagem por memorização utilizando um $\mathscr{A}$-AEF

A aprendizagem por memorização é um exemplo trivial de técnica de aprendizagem de máquina em que exemplos positivos de um conceito são simplesmente armazenados e posteriormente recuperados na classificação de novos exemplos. O único mecanismo

de generalização deste dispositivo é assumir que tudo que não for explicitamente informado não é um exemplo do conceito a ser aprendido. Mostraremos agora um $\mathscr{A}$-AEF capaz de realizar aprendizagem por memorização, que pode ser visto também como versão ligeiramente modificada do coletor de nomes proposto em (NETO, 1993). Neste exemplo, temos um alfabeto $\Sigma = \{a, b, M\}$, com o símbolo M (de Memorizar) sendo reservado para ser sufixo de cadeias de $\{a, b\}^*$ que devem ser memorizadas (o M funciona como um sinal que indica, para o autômato, as cadeias que devem ser memorizadas). Portanto, para que uma cadeia $\alpha \in \{a, b\}^*$ seja aceita, o autômato precisa antes receber a cadeia de treinamento αM. Este exemplo ilustra também a capacidade do dispositivo adaptativo para tratar dependências de contexto [2]

A figura 7.2 mostra o mecanismo subjacente do autômato de estados finitos adaptativo $M = \langle Q, \Sigma, q_0, F, \delta, Q_\infty, \Gamma, \Pi \rangle$, no qual $Q = \{q_0, q_1, q_f\}$, $\Sigma = \{a, b, M\}$, $q_0 = q_0$, $F = \{q_f\}$, δ contém as transições mostradas graficamente na figura 7.2, $Q_\infty = q_n$, com $n \in \mathbb{N} \cup \{i, f\}$, $\Gamma = (\{+, -\} \times (Q_\infty \times (\Sigma \cup \{\epsilon\}) \times Q_\infty))$ e a função Π é mostrada abaixo, com q' e q'' na figura 7.3 sendo usados para indicar dois estados em $Q_\infty - Q$ (estados que não se encontram no conjunto corrente de estados). Uma implementação deste exemplo deverá ainda cuidar de alguns detalhes, como a manutenção das alterações causadas pelas ações adaptativas elementares entre uma execução e outra. Uma outra alternativa seria a utilização de separadores entre cadeias (e.g ";") e a utilização de um transdutor capaz de oferecer, para cada cadeia da seqüência, uma saída indicando se a mesmo pertence ou não ao conceito que está sendo aprendido. O exemplo *adaptree_core.spa* (anexo B), disponível no pacote AdapTools, implementa esta solução.

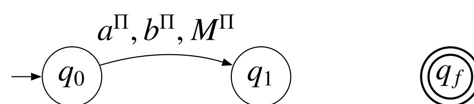


Figura 7.2: $\mathscr{A}$-AEF M que realiza aprendizagem por memorização (o estado q_f é propositadamente inalcançável)

As figuras 7.4 e 7.5 mostram o autômato M após a leitura dos símbolos a e M, respectivamente. A cadeia a, que seria rejeitada na configuração inicial (figura 7.2),

[2]Informalmente, a capacidade de um $\mathscr{A}$-AEF para tratar dependência de contexto pode ser verificada partindo-se da demonstração de equivalência entre autômatos adaptativos e máquinas de Turing (ROCHA; NETO, 2000a). Uma vez que pilhas podem ser simuladas através de autômatos adaptativos, sem usar pilhas (NETO, 1993), temos que a retirada das chamadas e retornos de submáquina não restringem o poder de expressão de um autômato adaptativo. Portanto, um $\mathscr{A}$-AEF , sendo basicamente um autômato adaptativo sem a pilha de controle de chamadas de submáquinas, tem o mesmo poder de expressão dos autômatos adaptativos, e conseqüentemente, das máquinas de Turing

$$\Pi(q_i, \alpha, q_j) = \{(-, q_i, \alpha, q_j), (+, q_i, \alpha, q'), (+, q', \beta, q'')\}, \forall\alpha \in \{a, b\}, q_i \neq q_j, \beta \in \Sigma$$
$$\Pi(q_i, M, q_j) = \{(+, q_i, \epsilon, q_f)\}, \forall q_i \neq q_j$$

Figura 7.3: Função Π para exemplo da aprendizagem por memorização

passa a ser aceita após a leitura da cadeia aM.

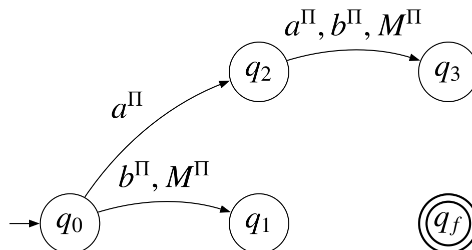


Figura 7.4: M Após a leitura do símbolo a

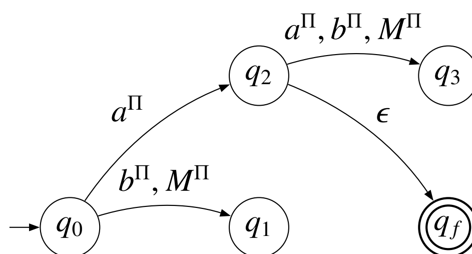


Figura 7.5: M Após a leitura da cadeia aM

7.4 Conclusão

O principal objetivo da introdução do mecanismo adaptativo aqui apresentado é auxiliar na divulgação do conceito de adaptatividade utilizando uma linguagem comumente encontrada em livros-texto na área de teoria da computação. O dispositivo foi criado inicialmente com uma finalidade didática. Como algumas das aplicações já sugeridas para autômatos adaptativos não utilizam a pilha do mecanismo subjacente, é razoável supor que os autômatos de estados finitos adaptativos também possam ser aplicados na prática. Uma sugestão interessante para trabalhos futuros é a reformulação do dispositivo adaptativo aqui apresentado usando a proposta apresentada na seção 2.2.

Analisando a demonstração do poder de expressão dos autômatos adaptativos apresentada em (ROCHA; NETO, 2000a), que não utiliza os recursos que diferenciam um autômato de estados finitos de um autômato de pilha estruturado, podemos intuir que

os autômatos de estados finitos adaptativos também devem ter poder de máquina de Turing. No entanto, uma demonstração formal de equivalência do dispositivo aqui proposto, com máquinas de Turing, precisará ser realizada no futuro (embora, para os fins didáticos aqui mencionados, esta equivalência não seja tão importante). Ressaltamos também que esta é apenas uma proposta inicial, que deve ser aprimorada no futuro, com a inclusão de demonstrações de propriedades fundamentais, capacidade expressiva, comportamento em relação à não-determinísmos e estudos sobre decidibilidade em relação aos problemas geralmente analisados em outros formalismos.

Parte III

Ferramentas

8 ADAPTOOLS

Neste capítulo apresentaremos o AdapTools [1], um ambiente computacional em que autômatos adaptativos podem ser implementados, depurados e experimentados. O núcleo deste sistema é uma máquina virtual que executa uma versão ligeiramente modificada de um autômato adaptativo. As modificações no formalismo original visam principalmente à simplificação e à uniformização no formato de especificação de transições internas, transições externas e ações adaptativas elementares. Com isto, todos os elementos do dispositivo passam a poder ser apresentados através de uma única tabela. O ambiente computacional inclui ainda recursos de depuração, como execução passo-a-passo, visualização de variáveis e pilhas, e animação gráfica, bem como mecanismos para controle de projetos, execução simultânea de múltiplos dispositivos (com comunicação entre si) e uma série de exemplos de autômatos.

A próxima seção apresenta a versão modificada de autômatos adaptativos utilizada no AdapTools. Em seguida, apresentamos algumas características da interface do sistema com o usuário. Na seção 8.3 mostraremos como rotinas semânticas podem ser atreladas a um autômato adaptativo. As possibilidades da integração entre diferentes autômatos na construção de sistemas mais complexos e a utilização da máquina virtual do AdapTools em sistemas externos são discutidas nas seções 8.4 e 8.5. A seção 8.7 aponta algumas direções e melhorias previstas para as futuras versões desta ferramenta.

8.1 Autômatos Adaptativos no AdapTools

O formato escolhido para representar um autômato adaptativo no AdapTools é uma tabela com 7 colunas: (1) cabeçalho (*head*), (2) estado de origem (*orig*), (3) símbolo de entrada (*inpu*), (4) estado de destino (*dest*), (5) empilha (*push*), (6) símbolo de saída (*outp*) e (7) ação adaptativa (*adap*). As linhas desta tabela são de 3 tipos básicos:

[1] Neste texto considera-se apenas a versão 1.4.2 do AdapTools

transição interna, transição externa (chamadas de sub-máquina e retorno) e ação adaptativa elementar. A primeira coluna desta tabela, o cabeçalho, contém o nome da sub-máquina ou da função adaptativa à qual o elemento (na linha) pertence. No caso de linhas referentes a ações adaptativas elementares, o nome da função adaptativa é precedida de um símbolo indicando o tipo da ação adaptativa elementar: ?, $-$ e $+$, para ações elementares de consulta, remoção e inserção, respectivamente. As colunas de estados de origem e destino, e símbolos de entrada e saída, carregam justamente o que os seus nomes indicam (para se utilizar o AdapTools na execução de simples transdutores de estados finitos, estas são as únicas colunas necessárias). Estas 4 colunas poderão conter também variáveis, geradores e parâmetros, no lugar de constantes, no caso das linhas que representam ações adaptativas elementares. Seguindo a proposta apresentada no capítulo 6, variáveis, geradores e parâmetros são implicitamente declarados, sendo identificados pelos prefixos ?, $*$ e %, respectivamente [2].

As demais colunas estão relacionadas com a especificação das camadas que estendem o transdutor de estados finitos para um autômato de pilha estruturado e autômato adaptativo. A coluna empilha contém o endereço (estado) de retorno de uma chamada de sub-máquina. Retornos de sub-máquina são identificados pela presença da palavra reservada *pop*, na coluna de estado de destino. Ações adaptativas são acopladas às transições através de última coluna. Os símbolos dessa coluna apresentam o formato $A.P$, em que A e P são nomes de funções adaptativas anteriores (A) e posteriores (P), ambos opcionais, seguidos ou não de uma seqüência, entre parênteses, de parâmetros, separados por vírgula. A palavra reservada *nop* deve ser inserida em todos os campos que não estejam sendo utilizados na especificação de um determinado dispositivo.

Ainda para evitar a utilização de estruturas adicionais, além da tabela, optamos por codificar os estados iniciais e finais do autômato, da seguinte maneira: a coluna empilha deve conter a palavra reservada *fin*, quando o estado de destino for um estado final (além disto, o estado de origem das transições de retorno de sub-máquina também é automaticamente identificado pela máquina virtual como um estado final). O estado inicial de uma sub-máquina será sempre aquele que se encontra na primeira linha cujo cabeçalho contém o nome da sub-máquina. A listagem abaixo descreve cada uma das palavras reservadas e construções especiais utilizadas no AdapTools:

nop Pode ser utilizado apenas nas colunas 5, 6 e 7 para indicar ausência de estado de

[2] Até a versão 1.4.2, o AdapTools adotava o prefixo ?*p* para parâmetros

retorno, símbolo de saída ou ação adaptativa.

eps Representa a cadeia vazia, ϵ.

pop Quando o estado-destino é *pop* temos um transição de retorno de sub-máquina, ou seja, o próximo estado será retirado (*pop*), em tempo de execução, da pilha de chamadas de sub-máquinas.

fin Pode ser colocado na coluna empilha para indicar que o estado de destino da transição é um estado final.

spc Usado na coluna de símbolo de entrada para representar os caracteres ASCII que, em geral, funcionam como separadores: espaço, salto de linha e tabulação.

?sta Variável especial que pode ser utilizada apenas em ações adaptativa elementares. Em tempo de execução, é instanciada com o valor do estado corrente do autômato.

?inp Variável especial instanciada com o valor do próximo símbolo da entrada.

As palavras reservadas seguintes podem ser utilizadas apenas na coluna de símbolo de entrada. Elas simplificam as especificações de determinados conjuntos de transições, o que, de outra forma, exigiria a inserção de várias linhas na tabela que define um autômato:

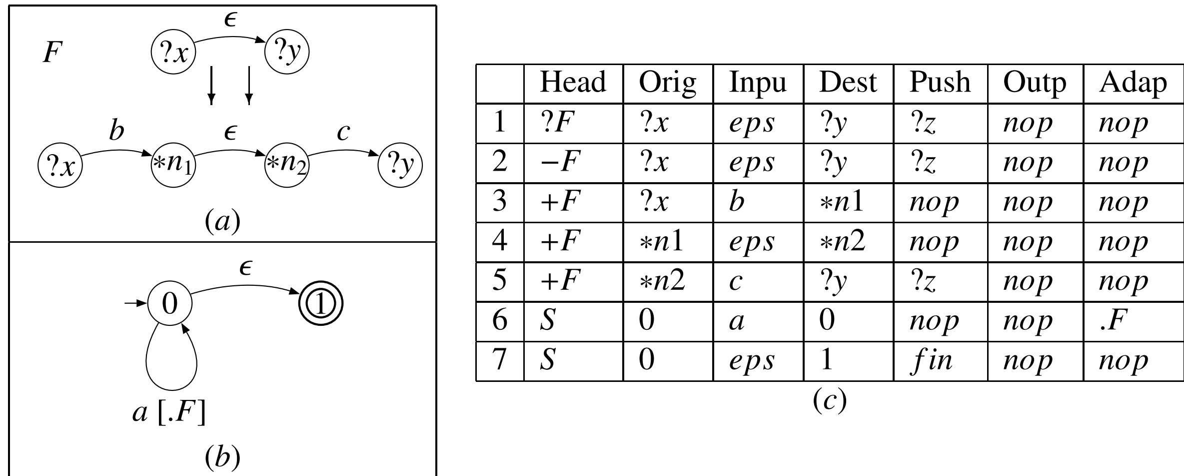
n..m Faixa de valores ASCII de n a m. Por exemplo, a presença do valor "*p..t*" na coluna de símbolo de entrada de uma transição indica que a mesma consome qualquer um dos símbolos p, q, r, s e t.

digit O mesmo que 0..9;

letter O mesmo que *a..z* e *A..Z*;

special Qualquer símbolo que não seja nem letra nem dígito.

other Qualquer símbolo que não possa ser consumido estando no estado de origem da transição (lembrando que deste estado podem, e geralmente estarão, saindo outras transições). Transições com este símbolo estão muitas vezes associadas a uma função adaptativa de tratamento de erro, através de uma ação adaptativa.



	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	$?F$	$?x$	eps	$?y$	$?z$	nop	nop
2	$-F$	$?x$	eps	$?y$	$?z$	nop	nop
3	$+F$	$?x$	b	$*n1$	nop	nop	nop
4	$+F$	$*n1$	eps	$*n2$	nop	nop	nop
5	$+F$	$*n2$	c	$?y$	$?z$	nop	nop
6	S	0	a	0	nop	nop	$.F$
7	S	0	eps	1	fin	nop	nop

Figura 8.1: Autômato adaptativo que reconhece $a^n b^n c^n$
(a) Mecanismo adaptativo (b) Mecanismo subjacente (c) Código objeto da máquina virtual do AdapTools

As figuras 8.1.(a) e 8.1.(b) mostram um autômato adaptativo para $a^n b^n c^n$ usando a representação gráfica sugerida na seção 6.2, ao lado da representação tabular, empregada no AdapTools (figura 8.1.(c)). A representação tabular deve ser interpretada da seguinte forma: na linha 6, temos a transição que consome uma seqüência de símbolos a. A última coluna desta linha indica que a função adaptativa F deve ser executada após $(.F)$ o consumo de cada símbolo a. A função adaptativa F realiza a busca de uma transição contendo o símbolo eps na coluna de entrada (transição em vazio), na linha 1 (ação elementar de consulta), remove esta transição (linha 2) e insere três novas transições, que consomem cadeias bc (linhas 3 e 5), além de manter uma transição vazia no autômato, estrategicamente posicionada para garantir o reconhecimento correto da linguagem (linha 4).

Um exemplo mais complexo de autômato adaptativo, implementado através do AdapTools, é apresentado no anexo D. Nesse exemplo tem-se uma camada adaptativa capaz de realizar uma cópia do autômato subjacente. É interessante notar, nesse exemplo, a utilização de passagem de parâmetros e de transições em vazio acopladas a funções adaptativas na obtenção de uma estrutura de controle de repetição. A execução da camada adaptativa consiste basicamente em uma busca em largura com detecção de laços. Uma versão mais simples, sem detecção de laços, também está disponível no pacote AdapTools.

Produções Iniciais

(1 , "B") : → 4
(2 , "B") : → 3
(3 , ")") : → 4
(1 , "(") : → 2 , $\mathscr{A}$(2,3,1)

Função Adaptativa

$\mathscr{A}$(i,j,n) =	{ k* , m* :
	+ [(k , "B") : → m]
	+ [(m , ")") : → j]
	+ [(i , "(") : → k , $\mathscr{A}$(k,m,i)]
	+ [(n , "(") : → i , $\mathscr{A}$(i,j,n)]
	+ [(n , "(") : → i]

Figura 8.2: Simulador de pilhas na versão original

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	S	1	B	4	fin	nop	nop
2	S	2	B	3	nop	nop	nop
3	S	3	)	4	nop	nop	nop
4	S	1	(	2	nop	nop	.A(2,3,1)
5	+A	*k	B	*m	nop	nop	nop
6	+A	*m	)	$\%_2$	nop	nop	nop
7	+A	$\%_1$	(	*k	nop	nop	.A(*k,*m,$\%_1$)
8	-A	$\%_3$	(	$\%_1$	nop	nop	.A($\%_1$,$\%_2$,$\%_3$)
9	+A	$\%_3$	(	$\%_1$	nop	nop	nop

Figura 8.3: Simulador de pilha no AdapTools

8.1.1 Diferenças em Relação à Definição Original

A operação do autômato adaptativo no AdapTools segue a proposta descrita no capítulo 6, no que tange à camada adaptativa. A camada subjacente, o autômato de pilha estruturado, funciona de maneira análoga à da definição apresentada em (NETO, 1993), com uma exceção: na versão atual, a devolução de símbolos para a cadeia de entrada só é possível através de rotinas semânticas. Reproduzimos na figura 8.2 um exemplo de autômato adaptativo, o simulador de pilha, apresentado em (NETO, 1993), fornecendo em seguida, na figura 8.3 o autômato equivalente no AdapTools.

8.1.2 Formato do Arquivo Gerado pelo AdapTools

O formato de um arquivo contendo o “código objeto” de um autômato adaptativo é basicamente um *dump* da tabela mostrada na interface gráfica do AdapTools, com o símbolo de ponto-e-vírgula sendo usado como separador de colunas. A primeira linha do arquivo deve conter a cadeia “[Version] 2”. Este recurso foi utilizado para permitir alterações no formato do arquivo em futuras versões, mantendo-se a compatibilidade com os arquivos anteriores. O AdapTools aceita também arquivos sem esta primeira linha, e sem as colunas de cabeçalho e chamadas de funções adaptativas, neste caso, o arquivo corresponde apenas a um autômato estruturado (sem a camada adaptativa).

8.2 Interface do Sistema

A figura 8.4 apresenta a tela principal do aplicativo, acrescida de marcadores (letras em azul) para os principais componentes de interface, que estão descritos a seguir. O principal componente é a tabela (figura 8.4.(a)) contendo o código do autômato adaptativo. As tarjas vermelha e lilás marcam, respectivamente, a transição e a ação adaptativa elementar (se for o caso) que estão sendo executadas a cada momento.

As três caixas de textos à direita da janela estão associadas à cadeia de entrada (figura 8.4.(d)), à cadeia de saída (figura 8.4.(e)), para os casos em que o autômato funciona como um transdutor; e a uma saída auxiliar de texto (figura 8.4.(f)). Esta saída auxiliar de texto é geralmente manipulada por rotinas semânticas, associadas às regras de transição do autômato, mas especificadas utilizando a linguagem de programação em que o AdapTools foi desenvolvido (ver seção 8.3). Uma cor diferente para o fundo da caixa de texto de entrada é utilizada para marcar a parte de cadeia já lida pelo autômato.

Os botões, mostrados nas figuras 8.4.(j) e 8.4.(k), são utilizados para iniciar, dar continuidade à execução (no modo passo a passo) e reiniciar a máquina virtual. A velocidade de execução pode ser controlada por uma barra de rolagem (figura 8.4.(c)), que quando arrastada até seu ponto mais inferior, coloca a máquina em execução passo a passo. A pilha de chamadas de sub-máquinas, os estados finais e o estado atual são apresentados através dos componentes (b), (g) e (h), respectivamente. Finalmente, o componente (i) é utilizado para mostrar o conteúdo de variáveis e geradores instanciados por ações elementares de consulta. Durante a execução de ações elementares de

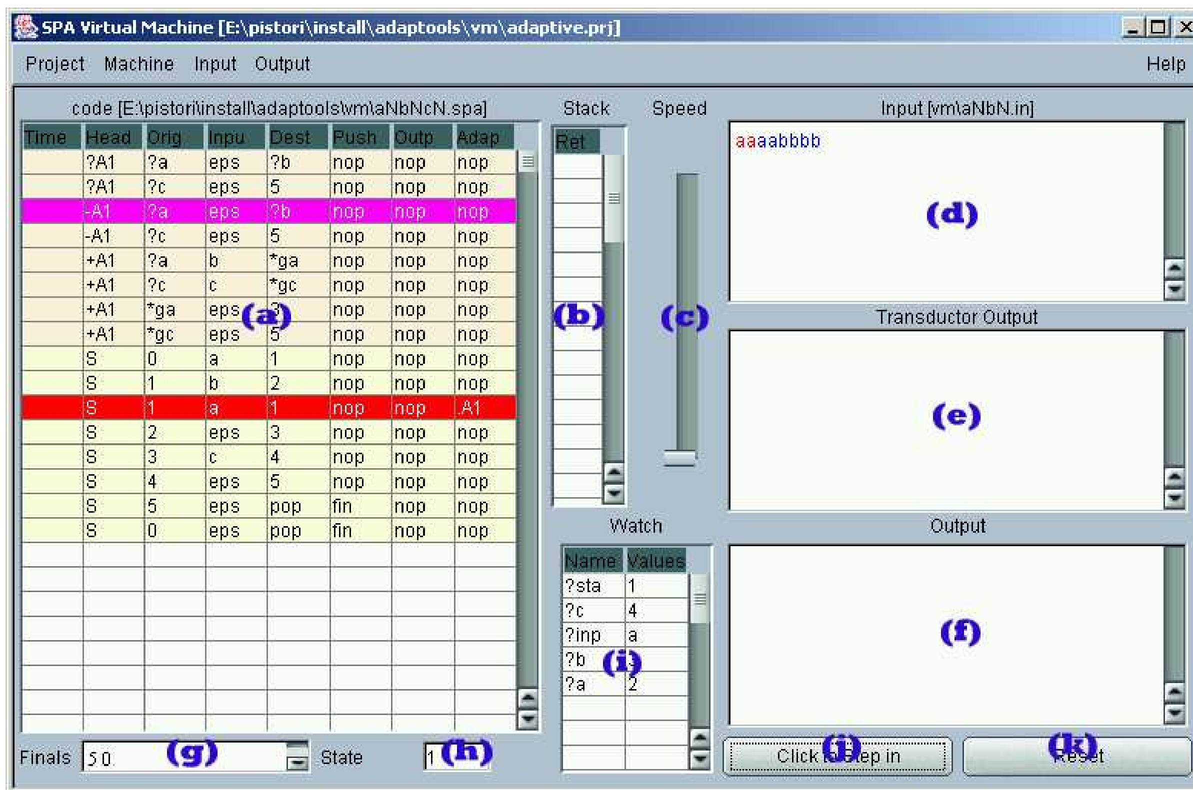


Figura 8.4: Janela principal do AdapTools

consulta, uma terceira tarja colorida é utilizada para mostrar as transições que satisfazem às restrições impostas.

A barra de menus agrega as operações usuais para manipulação de arquivos, que neste caso, incluem arquivos que armazenam a especificação de autômatos (menu *Machine*), cadeias de entrada (menu *Input*) e saída semântica (menu *Output*). O AdapTools oferece algumas opções para agregar diferentes arquivos em um único projeto, facilitando assim a sua manipulação. Arquivos de projetos possuem extensão "prj" e podem ser criados e recuperados através das opções do menu de projeto (menu *Project*). A utilização de projetos facilita sobremaneira o trabalho com sistemas que incluem múltiplas máquinas (ver próxima seção) e permitem ainda que um texto, de ajuda, seja associado ao projeto. Este arquivo de ajuda é automaticamente carregado e disponibilizado ao usuário através do menu de ajuda (menu *help* - na extremidade direita da barra de menus). O menu de ajuda oferece acesso também a um hipertexto contendo um tutorial sobre o AdapTools.

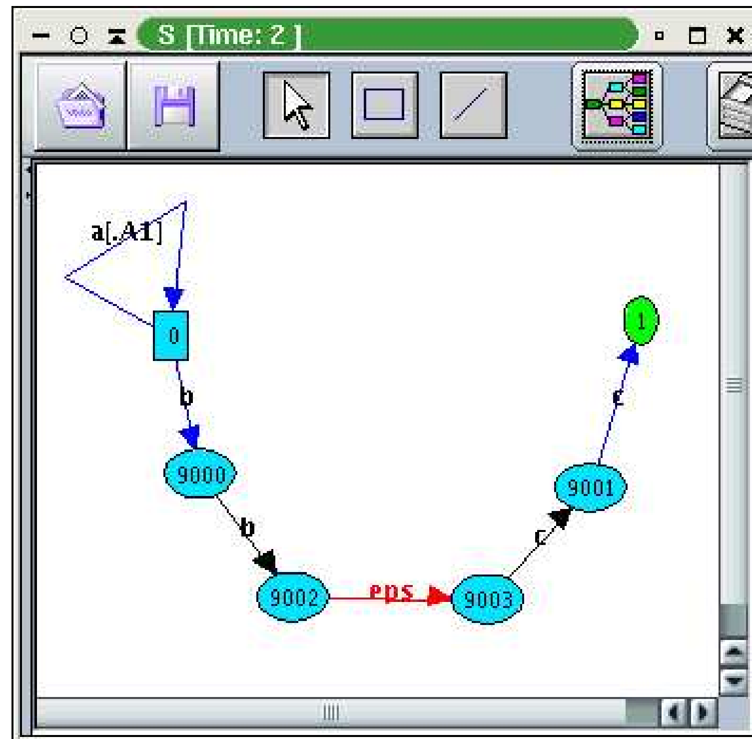


Figura 8.5: Módulo de animação gráfica do AdapTools

8.2.1 Visualização Gráfica

Usando o menu de opções (*options*) ou o clique invertido do mouse sobre alguma das transições mostradas na tabela com o código objeto, é possível ativar o módulo de visualização gráfica da AdapTools. Este módulo, construído a partir do software livre OpenJGraph [3], mostra graficamente o mecanismo subjacente do autômato adaptativo, animando as modificações sofridas durante sua execução. Um algoritmo de desenho automático de grafos, implementado no OpenJGraph, busca posicionar os vértices e arestas que representam graficamente o autômato, de forma que a visualização seja facilitada. Este algoritmo é inspirado em leis da física relacionadas com atração e repulsão entre corpos. Os vértices, neste algoritmo, modelam corpos que se repelem, ligados entre si por cordas elásticas (arestas), que contra-balanceiam a força de repulsão. O objetivo final do algoritmo é encontrar um ponto de equilíbrio para o conjunto de vértices (corpos), respeitando valores pré-definidos para as variáveis envolvidas no modelo (força de atração, elasticidades,etc). O módulo de animação permite ainda a interação do usuário, que pode movimentar os vértices, com a ajuda do mouse, buscando uma melhor visualização. A figura 8.5 mostra a janela de animação gráfica do

[3]Disponível gratuitamente em http://openjgraph.sourceforge.net/

AdapTools. Diferentes cores são utilizadas para destacar transições recém-inseridas (preto), recém-removidas (vermelho) e estados finais (verde).

8.3 Tratamento de Rotinas Semânticas

Rotinas semânticas podem ser introduzidas em um autômato adaptativo através da coluna de símbolo de saída (*outp*). Ao executar o autômato, a máquina virtual, antes de enviar o símbolo de saída para a cadeia de saída, faz uma chamada de um procedimento interno, pré-definido, chamado *execute*. Este procedimento deve analisar o símbolo de saída e retornar para a máquina virtual uma cadeia qualquer de símbolos (que pode ser vazia), a ser impressa na janela de saída auxiliar de texto do AdapTools. O procedimento *execute* pode ser substituído por outro, através da manipulação de códigos-fonte em Java e a recompilação do sistema. Para auxiliar o processo de escrita de rotinas semânticas, o pacote AdapTools dispõe de uma classe denominada *Adaptools.vm.Semantics*, que pode ser especializada para a construção de novos módulos para tratamento de rotinas semânticas. Estes módulos podem, internamente, utilizar todos os recursos oferecidos pela linguagem Java (um dos exemplos do AdapTools usa uma saída sonora, em lugar de textual). O AdapTools implementa algumas rotinas semânticas padrão, que podem ser utilizadas sem a necessidade de recompilação de código. Uma listagem dessas rotinas semânticas, juntamente com detalhes sobre como implementar novas rotinas, podem ser encontrados no anexo C.

8.4 Comunicação entre Máquinas Virtuais

A saída de uma rotina semântica (figura 8.4.(f)) de uma máquina *A* pode ser redirecionada para uma outra máquina *B* através da opção de conexão (*Connect*) do menu *input* na máquina *B*. Este recurso pode ser usado, por exemplo, na construção de compiladores, quando a análise léxica e sintática são executadas por módulos distintos. O tipo básico de dados transferidos de uma máquina para outra é um *token*, formado por duas cadeias de caracteres, representando seu nome (*token.label*) e seu conteúdo (*token.value*). Os *tokens* devem ser gerados e lidos através de rotinas semânticas, que, como foi visto na seção anterior, requerem um trabalho adicional por parte do usuário. Para facilitar um pouco esta tarefa, o AdapTools já inclui algumas rotinas semânticas que facilitam o trabalho com *tokens* (sem necessidade de se abrir os fontes do sistema).

Estas rotinas também encontram-se no anexo C.

8.5 Integração com Outros Sistemas

A máquina virtual do AdapTools pode ser executada sem os recursos gráficos apresentados nas seções anteriores, facilitando assim a integração com sistemas externos. Nos casos em que o sistema externo estiver sendo desenvolvido em Java, esta integração é trivial, e consiste basicamente em incluir e utilizar a classe *Runner* do pacote AdapTools no sistema em questão. Para sistemas desenvolvidos em outras linguagens, não existe ainda um mecanismo para facilitar esta integração. No entanto, é possível realizá-la, ainda que de forma rudimentar, utilizando uma versão do AdapTools que pode ser executada através da linha de comando do sistema operacional. Esta versão pode ser integrada a outros sistemas utilizando-se os recursos do próprio sistema operacional para a comunicação entre processos (e.g. *pipes* no Linux).

8.6 Arquitetura do Sistema

Destacam-se agora alguns elementos do projeto do AdapTools que poderão ser úteis a futuros desenvolvedores ou àqueles que tenham a intenção de reutilizar, total ou parcialmente, os códigos-fonte do AdapTools em outros projetos. Uma característica fundamental deste projeto é a separação entre: (1) a máquina virtual que implementa o mecanismo subjacente (*Kernel.java*), (2) o mecanismo adaptativo (*Adapter.java*) e (3) os tipos de dados abstratos de apoio à execução da máquina virtual, como por exemplo, a tabela que armazena o autômato, a pilha de chamada de sub-máquinas e o estado corrente, entre outros (*Vmds.java*[4]).

O módulo *Vmds.java* é, basicamente, uma interface (no sentido utilizado em programação em java: *keyword interface*), que deve ser implementada para oferecer as estruturas de dados requeridas pelos módulos *Adapter.java* e *Kernel.java*. Duas implementações são disponibilizadas no pacote. A primeira (*Viewer.java*) utiliza os próprios componentes visuais do pacote Swing e implementa a interface gráfica com o usuário . A segunda (*VmdlImpl.java*) oferece as estruturas de dados utilizadas pelo módulo textual (linha de comando) do AdapTools (*Runner.java*).

[4] *Virtual Machine Data Structures*

8.7 Considerações sobre a Máquina Virtual do AdapTools

O formato tabular utilizado na especificação de um autômato adaptativo não foi, a principio, criado para ser utilizado por um usuário final de tecnologia adaptativa, mas para ser gerado automaticamente por um módulo que permitisse a especificação de autômatos em uma linguagem mais acessível (possivelmente gráfica). No entanto, como na versão a que se refere esta tese, o formato tabular é o único instrumento disponível para inserção e manipulação de especificações de autômatos do AdapTools, acabamos por permitir algumas simplificações que encobriram ligeiramente as características da máquina virtual. A principal delas é a sintaxe da coluna de ação adaptativa. Devido à presença dessa coluna, a máquina virtual do AdapTools é forçada a implementar internamente algums mecanismos de análise léxica e semântica, para interpretar seu conteúdo. Em versões posteriores, quando estiver disponível um mecanismo de mais alto nível para a especificação dos autômatos, a máquina virtual deverá ser reprojetada.

Além da questão acima citada, o projeto da máquina virtual implementada não considerou requisitos de desempenho. Uma importante meta para as próximas versões do AdapTools é a construção de uma máquina virtual otimizada, possivelmente com implementação em linguagem de máquina (e não apenas em byte-codes Java) e considerando questões de desempenho da execução das funções adaptativas, em autômatos grandes, levantadas no capítulo 6.

A possibilidade de se realizar a escrita de símbolos na cadeia de entrada, um mecanismo bastante útil na especificação de determinados problemas no projeto de compiladores (NETO, 1993), e no processamento de linguagens naturais (NETO; MORAES, 2002), também é uma questão que deve ser resolvida nas próximas implementações do AdapTools. A adição deste mecanismo deverá ser precedida, provavelmente, de uma modificação na estrutura tabular dos autômatos adaptativos. Uma primeira sugestão seria a adição de uma nova coluna para especificar o destino de cada símbolo da saída. Especial cuidado deverá ser tomado em relação à comunicação inter-máquinas, pois a entrada de uma máquina pode estar conectada à saída de uma segunda máquina. Esta segunda máquina é executada de forma concorrente com a primeira (cada uma em seu *thread*), e portanto, tratamentos de concorrência entre processos mais sofisticados deverão ser implementados. Uma outra sugestão para trabalho futuro seria justamente

permitir a execução distribuída e concorrente da máquina virtual do AdapTools.

Também seria muito interessante oferecer ao usuário final uma maneira mais simples de trabalhar com rotinas semânticas, que não exija a recompilação de códigos-fonte. A primeira alternativa a ser investigada poderia ser a criação de um mecanismo que permita a inserção de código Java diretamente na especificação do autômato, ou eventualmente em uma gramática que possa ser convertida para autômato adaptativo (algo parecido com o *yacc* para a linguagem C (LEVINE; MASON; BROWN, 1992)). A segunda envolve a implementação de conjuntos mínimos, mas expressivos, de rotinas semânticas básicas (inserção em pilha, remoção de pilha, escrita na saída padrão, etc) que possam ser utilizadas na solução de problemas em um determinado domínio (e.g. construção de compiladores). A versão atual já apresenta algumas rotinas semânticas básicas, no entanto, elas foram projetadas para resolver problemas bastante específicos, como a implementação do compilador Wirth-AdapTools que será apresentado no capítulo 11. Um estudo mais profundo sobre o conjunto de rotinas semânticas "ideal", para determinados domínios de aplicação, deverá ser realizado futuramente.

Por último, novos módulos deverão ser acrescentados ao AdapTools para permitir a manipulação de outros tipos de dispositivos adaptativos. Esta generalização poderia começar com a criação de um formato tabular para a especificação de dispositivos adaptativos, em que as colunas referentes ao mecanismo subjacente pudessem ser definidas dinamicamente, em função do dispositivo adaptativo específico a ser implementado. Uma outra alternativa seria a criação de compiladores capazes de converter um dispositivo adaptativo específico em um autômato adaptativo que pudesse ser executado pelo AdapTools.

8.8 Conclusões

Apresentamos neste trabalho uma ferramenta para a área da tecnologia adaptativa que pode ser utilizada, pelo menos de quatro maneiras diferentes: (1) no projeto e desenvolvimento de soluções baseadas em autômatos adaptativos, (2) como uma ferramenta educacional para o ensino da tecnologia adaptativa, (3) na construção de ambientes de apoio ao aprendizado em áreas como construção de compiladores e teoria dos autômatos (usando apenas o mecanismo subjacente do autômato adaptativo) e (4)

como passo inicial na implementação de outros dispositivos adaptativos, como a árvore de decisão adaptativa apresentada no capítulo 5.

Como continuidade deste trabalho visualizamos pelo menos quatro objetivos importantes: (1) otimização da máquina virtual com implementações em linguagens de máquina para diferentes plataformas, (2) aprimoramento da execução simultânea e comunicação entre diferentes autômatos adaptativos, com implementação distribuída e generalizando os mecanismos de leitura e escrita de símbolos (e.g. permissão para escrita na cadeia de entrada), (3) criação de maneiras mais amigáveis para o tratamento semântico e (4) generalização do AdapTools para permitir a utilização de outros dispositivos adaptativos, além dos autômatos.

O pacote AdapTools pode ser obtido no *site* do Laboratório de Linguagens e Tecnologias Adaptativas da Universidade de São Paulo (http://www.pcs.usp.br/˜lta) ou diretamente no endereço http://www.ucdbnet.com.br/adaptools. A página do AdapTools oferece ainda diversos exemplos de autômatos, tutoriais, listas de pendências, documentação do sistema e indicações de como participar no desenvolvimento deste pacote, que é distribuído como software livre.

9 INTEGRAÇÃO DE DISPOSITIVOS ADAPTATIVOS, DE APRENDIZAGEM DE MÁQUINA E DE VISÃO COMPUTACIONAL

Para aplicar a tecnologia adaptativa a problemas relacionados com a visão computacional, modificamos, acrescentamos novos módulos e integramos três pacotes existentes: (1) um para o processamento digital de imagens, (2) outro com bibliotecas para a implementação de algoritmos de aprendizagem computacional e (3) um último para captura de imagens em tempo real. O resultado desta integração é um conjunto de módulos que facilitam a criação de sistemas com duas características fundamentais: (1) possuem interfaces cuja entrada consiste em sinais visuais, capturados através de uma *webcam*, e (2) precisam ser treinados para responder corretamente aos sinais visuais. Dois exemplos de sistemas deste tipo são apresentados no capítulo 10.

Na próxima seção descreveremos cada um destes pacotes e sua principais funcionalidades. Na seção 9.2 detalharemos os resultados desta integração, bem como os problemas e as soluções encontradas, para que os pacotes pudessem ser utilizados em conjunto. Conclusões e sugestões de melhorias para futuras versões são apresentadas na última seção deste capítulo.

9.1 Descrição dos Pacotes Utilizados

Constatamos durante o desenvolvimento deste projeto que há disponibilidade de pacotes de excelente qualidade, escritos em Java, com código aberto e gratuitos, tanto na área de visão computacional, quanto na de aprendizagem de máquina. Embora predominem códigos abertos em C e C++, a portabilidade da linguagem C está restrita ao "núcleo ANSI", o que torna-se um problema quando os produtos a serem desenvolvidos envolvem dispositivos não-convencionais e interfaces gráficas (PISTORI, 2000). Considerando que o baixo custo dos produtos finais é uma de nossas metas, e que a portabilidade é uma ferramenta indireta importante para a redução de custos, (livre escolha de produtos, aumento na concorrência, etc) optamos pela utilização exclusiva de soluções baseadas em Java. Descreveremos a seguir cada um dos três pacotes utilizados.

9.1.1 Processamento Digital de Imagens

Para implementar e testar técnicas de visão computacional e de processamento digital de imagens, utilizamos o pacote ImageJ, que é uma versão multiplataforma, ainda em desenvolvimento, do software NIH Image, para Macintosh. Entre os recursos oferecidos pelo pacote destacamos a disponibilidade, com programas-fonte abertos, de diversos algoritmos para: manipulação dos mais variados formatos de arquivo de imagens, detecção de bordas, melhoria de imagens, cálculos diversos (áreas, médias, centróides) e operações morfológicas. Esse software disponibiliza também um ambiente gráfico que simplifica a utilização de tais recursos, além de permitir a extensão através de *plugins* escritos em Java. Um outro fator importante para a sua escolha é a existência de uma grande comunidade de programadores trabalhando em seu desenvolvimento, com novos plugins sendo disponibilizados freqüentemente. Um desses plugins, o *Hough-Circles*, foi desenvolvido pelo nosso grupo durante a execução deste projeto e está disponível na página do ImageJ [1].

[1] http://rsb.info.nih.gov/ij/

9.1.2 Aprendizagem de Máquina

O segundo pacote utilizado neste projeto foi o WEKA[2] (Waikato Environment for Knowledge Analysis) (WITTEN; FRANK, 2000; CUNNINGHAM; HOLMES, 1999), também escrito em Java e com programas-fonte abertos. O WEKA é um ambiente bastante utilizado em pesquisas na área de aprendizagem de máquina, pois oferece diversos componentes que facilitam a implementação de classificadores e agrupadores (*clustering tools*). Além disto, esse ambiente permite que novos algoritmos sejam comparados a outros algoritmos já consolidados na área de aprendizagem, como é o caso dos algoritmos C4.5, Backpropagation, KNN e naiveBayes, entre outros. Podemos com esse pacote obter facilmente resultados estatísticos comparativos da execução simultânea de diversos programas de aprendizagem em domínios variados, tais como o reconhecimento de caracteres, o reconhecimento de imagens e o diagnóstico médico.

9.1.3 Tratamento de Dispositivos de Captura de Imagens

O *Java Media Framework* [3] é um pacote para captura de sinais temporais em tempo real. Este pacote permite que programas em Java possam acessar, por exemplo, imagens capturadas através de uma *webcam* acoplada a um computador pessoal. O software abstrai detalhes das arquiteturas e interfaces de diferentes dispositivos de captura de imagem, facilitando assim a portabilidade dos aplicativos.

9.2 Desenvolvimento do Pacote Integrado

A figura 9.1 apresenta, utilizando uma notação gráfica similar à de um diagrama de fluxo de dados, a arquitetura básica de um sistema guiado por sinais visuais e aprendizagem. A linhas pontilhadas resumem o fluxo de informação relacionado com o treinamento do sistema, que acontece da seguinte forma: uma imagem do sinal (e.g. uma mão humana realizando o sinal manual para a letra *p*) a ser aprendido é capturado através de um *webcam*, com auxílio do JMF, juntamente com uma classificação para o sinal, que é fornecida através do teclado ou mouse (e.g. digitação da letra *p*). A imagem é transmitida, após ser devidamente convertida (Conversor J-I), para o módulo de processamento digital de sinais (ImageJ). Neste módulo são realizadas transformações

[2] http://www.cs.waikato.ac.nz/ ml/weka/
[3] http://java.sun.com/products/java-media/jmf/

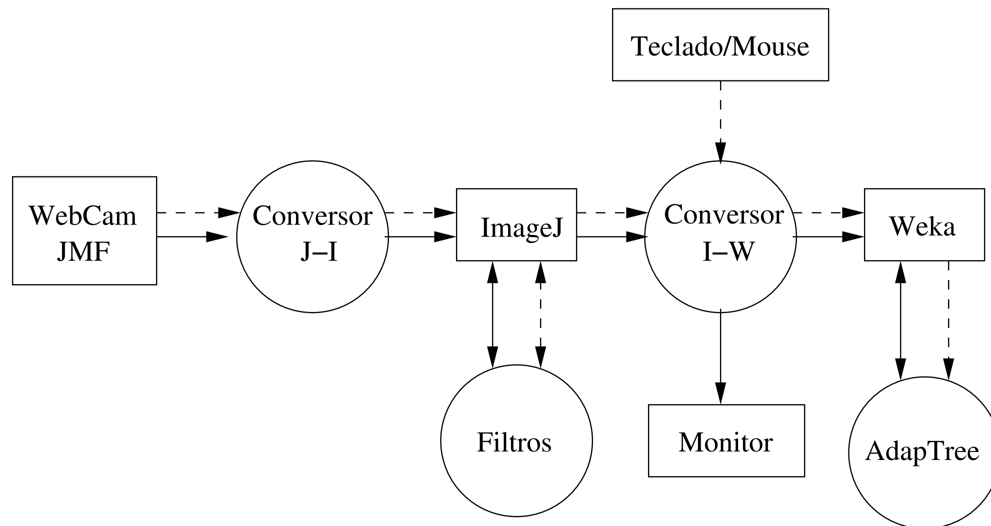


Figura 9.1: Sistema guiado por sinais visuais

na imagem que podem envolver filtros adicionais (Filtros), indisponíveis no pacote ImageJ. O resultado desta transformação é convertido e repassado, juntamente com a classificação do sinal, para o módulo de aprendizagem (Weka), que aciona o algoritmo de aprendizagem, neste caso, o AdapTree-E.

Depois que os sinais são aprendidos, o sistema entra no modo de funcionamento representado pelas linhas cheias da figura 9.1. Neste modo, a entrada consiste apenas das imagens capturadas pela *webcam* (sem as informações de treinamento). As imagens são processadas através dos mesmos filtros utilizados no treinamento e as informações resultantes são passadas para o módulo de aprendizagem. O módulo de aprendizagem deve agora oferecer uma resposta à entrada, com base no modelo abstraído a partir dos exemplos fornecidos anteriormente. Esta resposta é apresentada ao usuário através de algum dispositivo de saída, como por exemplo, o monitor do computador.

Nosso trabalho incluiu o desenvolvimento dos módulo de conversão J-I (JMF para ImageJ) e I-W (ImageJ para Weka), além é claro, da implementação do mecanismo de aprendizagem, AdapTree-E, e de filtros específicos para as duas aplicações apresentadas no capítulo 10. O resultado é um grupo de classes, escritas em Java, que podem ser especializadas ou modificadas na implementação de outros sistemas com características semelhantes.

9.3 Conclusões

Apresentamos neste capítulo um conjunto de pacotes e bibliotecas, alguns desenvolvidos por nós, outros disponíveis gratuitamente na Internet, que podem ser utilizados na implementação de sistemas guiados por sinais visuais. A integração dos pacotes pré-existentes não foi uma tarefa trivial, uma vez que, além de complexos, estes pacotes não foram projetados para trabalhar em conjunto. Um objetivo para o futuro é a construção de um ferramenta com interface gráfica que permita mais facilmente a utilização destes pacotes e bibliotecas, que hoje é feita basicamente através da alteração de programas-fonte escritos em Java. Também seria interessante buscar uma alternativa ao pacote JMF, que embora gratuito, não é livre (os programas-fonte não estão disponíveis), o que cria uma situação de dependência em relação ao seu desenvolvedor (neste caso, a SUN Microsystems).

A tecnologia adaptativa é incorporada aos sistemas guiados por sinais visuais através do AdapTree-E, que foi implementado em Java, utilizando bibliotecas disponibilizadas pelo Weka. Uma outra sugestão para novos trabalhos é a integração do AdapTools com as ferramentas aqui apresentadas. Desta forma, experimentos com outros dispositivos adaptativos, além do AdapTree-E, poderiam ser realizados no futuro.

Parte IV

Aplicações

10 PROCESSAMENTO DIGITAL DE IMAGENS E APRENDIZAGEM DE MÁQUINA

Neste capítulo são apresentadas algumas aplicações desenvolvidas para solução de problemas relacionados com processamento digital de imagens e aprendizagem de máquina. A principal ferramenta da tecnologia adaptativa, utilizada nestas aplicações, foi a árvore de decisão adaptativa, descrita no capítulo 5.

10.1 Interação Homem-Máquina através de Línguas de Sinais

O último senso brasileiro, realizado pelo IBGE [1], em 2000, indica um número de 166.000 cidadãos brasileiros surdos. Este número cresceria muito se considerássemos também pessoas com problemas severos de audição. A língua de sinais mais utilizada por essa comunidade é a LIBRAS, que apenas em 2002 passou a ser oficialmente reconhecida como meio legal de comunicação [2]. A mesma lei que reconhece a língua LIBRAS determina que professores, educadores especiais e fonoaudiólogos sejam treinados na utilização desta língua. O atraso no reconhecimento da língua de sinais brasileira é reflexo de um intenso e longo confronto entre defensores do oralismo e do gestualismo. Oralistas argumentam que pessoas surdas deveriam aprender, a qualquer custo, a língua portuguesa falada, para facilitar a sua inserção na sociedade não-surda. Já os gestualistas defendem o comportamento bi-lingüístico, com a língua de sinais sendo a primeira língua do surdo. O domínio da visão oralista, até recentemente (final

[1] Instituto Brasileiro de Geografia e Estatística

[2] Lei federal número 10.436 de 24 de abril de 2002

dos anos 80), levou até a algumas atitudes extremas em determinados países, como por exemplo, a proibição da utilização da língua de sinais na educação infantil (BRAFFORT, 2001). No entanto, três importante fatores estão mudando gradualmente este cenário, em favor da visão bi-lingüística: (1) o crescente reconhecimento da surdez como uma característica humana e não uma doença, (2) o reconhecimento da riqueza e da identidade da cultura da comunidade surda, o que inclui a língua LIBRAS, e (3) a existência de diversos estudos científicos demonstrando que a comunicação por sinais é essencial para o desenvolvimento integral de uma criança surda (SCHIRMER, 1994).

Línguas de sinais naturais não são meras transcrições de línguas faladas. Ao contrário, são sistemas com estrutura própria, em constante evolução, e tão expressivos quanto qualquer outra língua. Suas componentes léxicas, sintáticas e semânticas contam com recursos não encontrados em línguas orais, como espacialidade e iconicidade (BRAFFORT, 2001). Assim como acontece com as línguas faladas, línguas de sinais tendem a ser bem mais confortáveis, flexíveis e expressivas que suas correspondentes línguas escritas. Pessoas surdas podem se comunicar por sinais muito mais rapidamente do que através da escrita manual ou da digitação (SCHUMEYER; HEREDIA; BARNER, 1997). Portanto, a grande maioria dos argumentos que justificam pesquisas na área de tecnologias computacionais relacionadas com a fala, também se aplicam às línguas de sinais. Além disso, como as línguas de sinais não são universais, podendo inclusive possuir diversos dialetos dentro de um mesmo país, a tradução entre diferentes línguas de sinais é também um interessante alvo para pesquisas: as estimativas sobre a quantidade de línguas de sinais existentes variam de 4000 a 20000 (WOLL; SUTTON-SPENCE; ELTON, 2001).

Características específicas das línguas gestuais tornam impossível a completa reutilização de métodos e algoritmos desenvolvidos no domínio do reconhecimento da fala. Infelizmente, tais características são geralmente subestimadas por cientistas e engenheiros de computação que usufruem muito pouco das possibilidades de cooperação com a comunidade surda e com especialistas em língua de sinais (BRAFFORT, 2001).

Nesta seção apresentaremos um protótipo de um editor de texto que pode ser treinado para compreender sinais do alfabeto LIBRAS correspondentes às letras latinas. Os sinais são capturados por uma câmera digital e processados em um ambiente computacional que permite a fácil integração entre técnicas de aprendizagem de máquina e

processamento digital de sinais. Este editor pode ser utilizado tanto na experimentação de diferentes técnicas de aprendizagem de máquina quanto na captura e catalogação de imagens que poderão futuramente formar um banco de dados de treinamento, similar ao RVL-SLLL (MARTíNEZ et al., 2002) para língua americana de sinais. Embora ainda bastante rudimentar, este editor representa o primeiro passo na construção de um conversor LIBRAS-Português que poderá ser futuramente utilizado como um $front - end$ de aplicações mais complexas, como tradutores LIBRAS-Voz e sistemas com interfaces baseadas em LIBRAS. Nesta tese, o mecanismo de aprendizagem experimentado foi justamente o das árvores de decisão adaptativas, apresentadas no capítulo 5

Apresentamos a seguir um resumo de trabalhos correlatos na área de reconhecimento automático da língua de sinais, seguido de uma breve introdução à língua de sinais brasileira. As seções 10.1.3 e 10.1.4 descrevem com mais detalhes o protótipo de nosso editor LIBRAS e alguns experimentos realizados, envolvendo árvores de decisão adaptativas. Na última seção oferecemos uma análise dos resultados e possíveis trabalhos futuros.

10.1.1 Trabalhos Relacionados

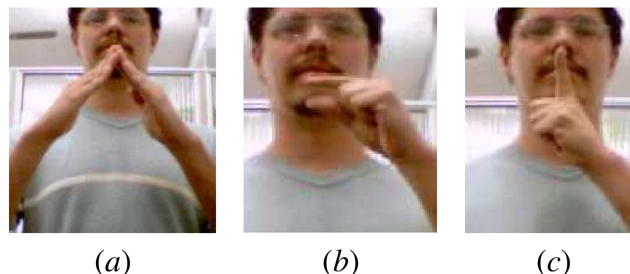
Embora existam alguns trabalhos recentes em reconhecimento de movimentos de cabeça (ERDEM; SCLAROFF, 2002), muito importantes em comunicação por sinais, a grande maioria dos trabalhos se concentra nos movimentos e formas da mão. Além da importância fundamental nas línguas de sinais para surdos, sistemas para rastreamento (*tracking*) das mãos humanas possuem aplicações em interação homem-máquina, realidade virtual e compressão de sinais, que incluem jogos controlados por movimentos de mão (FREEMAN et al., 1996), aparelhos de televisão acionados por sinais manuais capturados através de filmadoras (FREEMAN et al., 1996), compressão de sinais para vídeo-conferência (SCHUMEYER; HEREDIA; BARNER, 1997) e dispositivos que substituem o *mouse* na interação com computadores pessoais.

As duas principais classes de sistemas para rastreamento da mão são: (1) as baseadas em $data - gloves$, que requerem a utilização de luvas especiais de realidade virtual com diversos sensores para detecção das posições dos punhos, mãos, pontas dos dedos e articulações (FANG et al., 2001) e (2) as baseadas em visão computacional, que trabalham sobre um fluxo contínuo de imagens contendo as mãos e obtidas através de filmadoras digitais (L.BRETZNER; I.LAPTEV; T.LINDEBERG, 2002; MARTIN; DE-

VIN; CROWLEY, 1998; AL-JARRAH; HALAWANI, 2001; STENGER; MENDONçA; CIPOLLA, 2001). O primeiro grupo oferece geralmente maior velocidade e precisão, no entanto, o segundo grupo tem recebido cada vez mais atenção devido aos recentes avanços na área de processamento digital de imagens e à alta disponibilidade e baixo custo das câmeras digitais, além, é claro, de serem sistemas não intrusivos (não requerem que o usuário "vista" um equipamento). Algumas técnicas baseadas em visão computacional se utilizam também de recursos intrusivos, porém bem mais acessíveis que as $data - gloves$, como luvas convencionais marcadas com cores, que facilitam a identificação de partes das mãos (DAVIS; SHAH, 1994). Também existem sistemas que usam câmeras que captam sinais infra-vermelhos para facilitar o reconhecimento do corpo humano (SATO; KOBAYASHI; KOIKE, 2000). No entanto, a baixa disponibilidade e o alto custo das câmeras de infra-vermelho, quando comparado às câmeras digitais comuns, podem não justificar o ganho de precisão obtido.

Trabalhos específicos visando à construção de sistemas de reconhecimento de línguas de sinais para surdos já estão disponíveis para diversas línguas de sinais, como a australiana, a chinesa, a alemã, a árabe e a americana (FANG et al., 2001; AL-JARRAH; HALAWANI, 2001). A maioria desses sistemas incluem tipicamente quatro módulos: (1) segmentação, (2) extração de parâmetros, (3) reconhecimento de posturas e (4) reconhecimento de gestos.

Na fase de segmentação, a região das mãos é separada do fundo da imagem. Esta tarefa pode ser bastante complexa se não forem impostas restrições ao ambiente e se as mãos estiverem nuas. No entanto, bons resultados têm sido obtidos através de técnicas baseadas em esquemas de cores invariantes em relação à luminescência (KAPUSCINSKI; WYSOCKI, 2001), imagens de fundo previamente gravadas (MARTIN; DEVIN; CROWLEY, 1998) e cálculo das diferenças entre duas imagens subseqüentes (MARTIN; DEVIN; CROWLEY, 1998) . A extração de parâmetros reduz o espaço de busca, abstraindo da imagem segmentada algumas características importantes para a distinção entre diferentes sinais. A importância de uma característica está diretamente relacionada com a forma em que a modelagem dos sinais manuais é efetuada, podendo incluir estimativas para a posição e os ângulos relativos entre as pontas dos dedos, para a posição e direção do centro da mão, para o contorno da mão e para os *momentos da imagem* (*image moments*) (FREEMAN et al., 1998; SATO; KOBAYASHI; KOIKE, 2000; AL-JARRAH; HALAWANI, 2001; DAVIS; SHAH, 1994).



(a) (b) (c)

Figura 10.1: Exemplos de sinais icônicos em LIBRAS
(a) Casa (b) Pequeno (c) Silêncio

O reconhecimento de posturas e gestos consiste na busca pelo melhor modelo de sinal que casa com os parâmetros extraídos. Posturas são sinais que não envolvem movimento, por isto, sua modelagem é bem mais simples que a dos gestos. Modelagem de gestos envolve informação temporal e análise de seqüências de imagens. A grande maioria dos trabalhos em reconhecimento de gestos utiliza técnicas adaptadas do reconhecimento da fala, como as baseadas em cadeias de Markov (FANG et al., 2001; KAPUSCINSKI; WYSOCKI, 2001). Técnicas de aprendizagem de máquina, principalmente as baseadas em redes-neurais (AL-JARRAH; HALAWANI, 2001), também estão sendo utilizadas tanto no reconhecimento de postura quanto no reconhecimento de gestos. Outras técnicas utilizadas no reconhecimento incluem a análise dos componentes principais (PCA - *principal component analysis*) (MARTIN; CROWLEY, 1997), o casamento de padrões elásticos em grafos (*elastic graph matching*) (TRIESCH; MALSBURG, 1996) e os filtros de Kalman (STENGER; MENDONçA; CIPOLLA, 2001).

10.1.2 Língua de Sinais Brasileira

A língua de sinais brasileira, LIBRAS, é uma língua complexa, estruturada e natural, usada no Brasil, e com origens na língua de sinais francesa (CAPOVILLA; RAPHAEL, 2001). Como acontece com as línguas faladas, a língua de sinais evolui naturalmente, buscando a maior eficiência possível no meio de comunicação disponível. Na língua de sinais, este meio inclui movimentos de dedos, mãos, braços, tronco e cabeça, no espaço localizado à frente do interlocutor. Um exemplo interessante de aproveitamento otimizado do meio de comunicação, em LIBRAS, é o uso de diferentes posições espaciais para designar diferentes sujeitos do discurso, que podem ser referenciados posteriormente, durante o diálogo, através do movimento de apontar com os dedos. Outra característica fundamental da LIBRAS, possível devido à sua natureza visual, é o uso

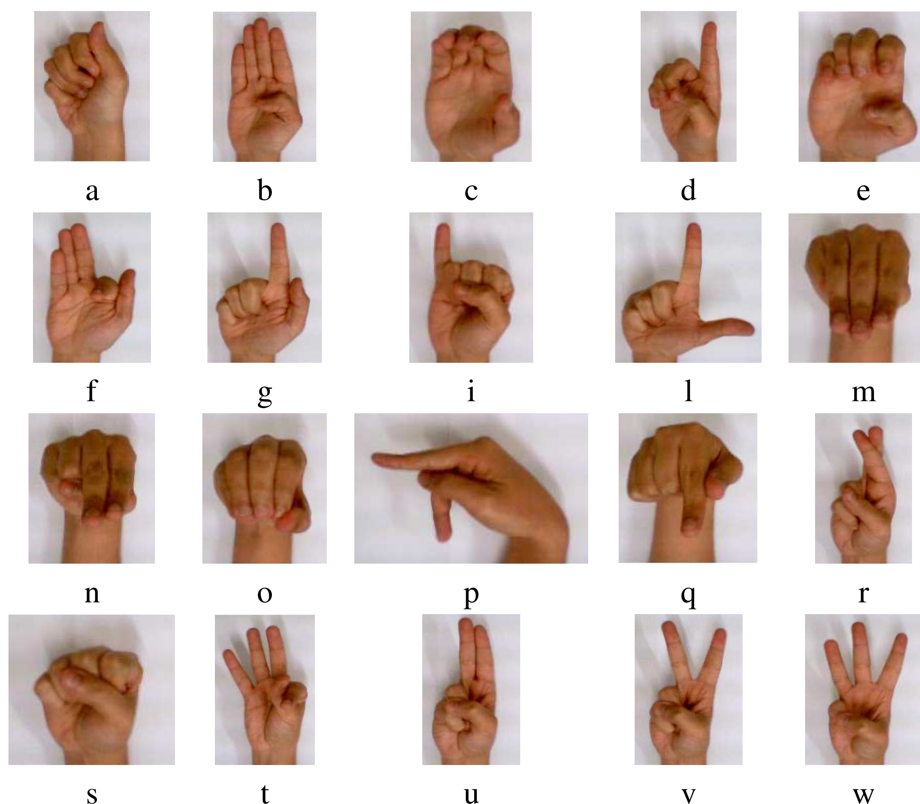


Figura 10.2: Alguns símbolos do alfabeto LIBRAS

de ícones com significado universal. A figura 10.1 mostra as posturas em LIBRAS que representam os conceitos de *casa*, *pequeno* e *silêncio*, respectivamente.

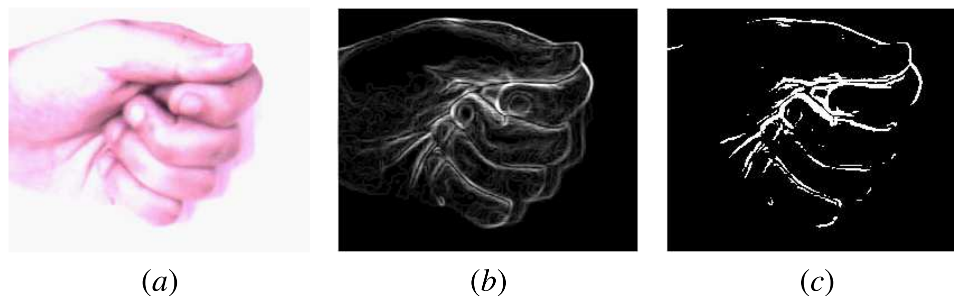
Embora a comunicação em LIBRAS inclua movimentos de cabeça, tronco, braços e outras partes do corpo; a forma da mão, sua posição em relação ao corpo do interlocutor e o movimento realizado por ela, funcionam muitas vezes como blocos elementares na construção de sentenças mais complexas (MARCATO; ROCHA; LIMA, 2000). A língua LIBRAS inclui um conjunto de 46 formas básicas para as mãos, que são também chamadas configurações. Este conjunto inclui 19 símbolos alfabéticos executados através de posturas (figura 10.2) e 6 símbolos alfabéticos, correspondentes as letras *h,j,k,x,y* e *z*, representados através de gestos. Em LIBRAS, o uso de seqüências de símbolos para formar palavras (*fingerspelling*) é restrito a alguns casos especiais, como acrônimos e nomes próprios. No entanto, símbolos alfabéticos aparecem com freqüência como componentes de outros sinais. A sentença *qual é o seu nome*, por exemplo, é expressa mostrando-se a letra *q* com as mãos próximas da boca (em referência ao pronome *qual*) e executando-se em seguida um movimento horizontal, da esquerda para a direita, na altura do peito, e com as mãos mostrando a letra *n* (em

referência ao substantivo *nome*). Informações adicionais sobre LIBRAS podem ser encontradas em (CAPOVILLA; RAPHAEL, 2001).

10.1.3 Desenvolvimento

Nós implementamos, em Java, um protótipo de um editor para símbolos alfabéticos da LIBRAS. A implementação foi executada utilizando as ferramentas descritas no capítulo 9. O protótipo trabalha da seguinte maneira: primeiramente, ocorre a fase de treinamento, quando o usuário deve executar os sinais manuais, com uma das mãos, e utilizar a outra para digitar a letra correspondente. Diversas imagens da mão são capturadas, até que uma tecla qualquer seja pressionada. Este procedimento é repetido, diversas vezes, para todas as letras desejadas: normalmente, quanto mais exemplos forem colhidos na fase de treinamento, maior será a precisão obtida na fase de reconhecimento. Quando uma tecla especial, reservada, for pressionada, o sistema induz a árvore de decisão adaptativa inicial, e o usuário pode então iniciar a edição do texto, quando apenas os sinais manuais passam a ser necessários. Cada sinal manual deve ser mantido por cerca de meio segundo para que o sistema tenha tempo de reconhecer três quadros consecutivos com a mesma letra (quando então a letra é efetivamente "digitada"). O módulo de aprendizagem pode ser acionado a qualquer momento (por exemplo, quando o sistema comete um erro de classificação), bastando para isto digitar a letra correspondente ao símbolo que está sendo mostrado para a câmera. Quando isto ocorre, a árvore de decisão adaptativa incorpora os novos exemplos de treinamento, conforme explicado no capítulo 5. Testes preliminares (usando a implementação de redes neurais do WEKA) indicam que a incorporação de novos exemplos com árvores de decisão adaptativas pode ser feita de maneira muitas vezes mais rápida do que com redes neurais artificiais tradicionais (que precisam ser totalmente retreinadas), no entanto, testes mais elaborados e formais devem ser realizados no futuro.

Além de um editor, bastante rudimentar, e do módulo de aprendizagem, o protótipo inclui ainda um módulo de processamento digital de imagens, responsável pela extração dos atributos a serem utilizados no reconhecimento dos sinais alfabéticos. O processamento digital de sinais é detalhado nas próximas seções, divididas em pré-processamento e extração de atributos.



(*a*) (*b*) (*c*)

Figura 10.3: Pré-processamento no protótipo LIBRAS
(a) Imagem original (b) Detecção de bordas usando Sobel (c) Binarização e inversão

10.1.3.1 Pré-processamento

A figura 10.3.(a) mostra a imagem original, com resolução de 180x160, capturada através de um *webcam* Creative^{TM}. Um fundo branco, juntamente com a utilização de distâncias pré-fixadas entre a câmera e a mão, foram utilizados para tornar desnecessária uma fase de rastreamento da região das mãos. Os *drivers* que acompanham a *webcam* oferecem alguns recursos de pré-processamento por hardware que foram utilizados na obtenção de uma imagem com baixo contraste e alto brilho, criando um efeito que facilita a extração de bordas. Estes ajustes foram obtidos experimentalmente, testando-se diversas configurações e analisando-se o resultado da extração de bordas em diferentes poses.

O primeiro passo do pré-processamento por software inclui a conversão da imagem colorida (no sistema RGB) para uma imagem em escalas de cinza ($grey - scale$) e a aplicação de um filtro de Sobel para detecção de bordas. Este é um filtro clássico em processamento digital de imagens, cuja descrição pode ser encontrada na maioria dos livros-texto da área, como por exemplo, em (GONZALEZ; WOODS, 2002), e que possui implementação disponível no pacote ImageJ. O resultado deste processamento é mostrado na figura 10.3.(b). Com as bordas detectadas, a imagem é binarizada (passa a ter apenas duas cores: preto e branco) utilizando a técnica de limiarização iterativa (*iterative thresholding*) proposta por Riddler e Calvard (RIDLER; CALVARD, 1978), e também implementada no ImageJ. A imagem final, resultante do pré-processamento, é mostrada na figura 10.3.(c).

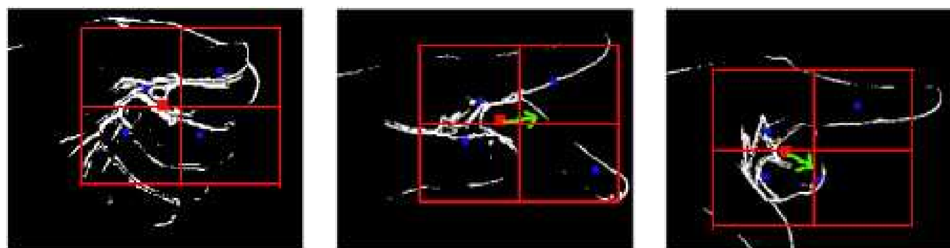


Figura 10.4: Parâmetros extraídos de três imagens diferentes

10.1.3.2 Extração de Atributos

A fase de extração de parâmetros é baseada em *momentos da imagem* (*image moments*) (FREEMAN et al., 1998) enriquecido por uma nova técnica que consiste em dividir a imagem em regiões retangulares de mesmo tamanho e calcular os parâmetros relacionados com o momento da imagem em cada uma dessas regiões. Esta estratégia fornece informações locais que facilitam a discriminação de determinados tipos de sinais manuais. O conceito de *momento* aqui empregado é inspirado na cinemática, e possui uma interpretação física quando consideramos os *pixels* brancos da imagem resultante da binarização como pontos em um espaço cartesiano. O conjunto destes pontos determina um objeto bidimensional para o qual podemos calcular, entre outras coisas, a massa total, o centro de massa e a direção. A figura 10.4 ilustra os 13 parâmetros que nosso método extrai para cada uma das imagens mostradas. O primeiro parâmetro corresponde a massa total (total de pontos da imagem binarizada). O reticulado, em vermelho, é posicionado no centro de massa da imagem e um novo centro de massa (quadrado vermelho), calculado sobre os pontos que estão dentro de reticulado, geram o segundo e terceiro parâmetro (coordenadas x e y do centro de massa). O próximo par de parâmetros captura a direção da imagem (seta verde) e pode ser calculado facilmente usando o desvio padrão dos pontos da imagem em relação ao centro de massa. Os retângulos em azul, que completam o conjunto de parâmetros extraídos, correspondem aos centros de massa de cada uma das quatro regiões do reticulado. A direção da massa em cada uma das regiões do reticulado não foi incluída neste primeiro estudo mas é uma informação que pode ser interessante na classificação do conjunto total de símbolos LIBRAS.

	AdapTree-E	C4.5	Backp.
PECC (300)	95.02%	95.23%	100%
PECC (Contexto)	69.10%	77.03%	92%
Tempo gasto no treinamento	0.27s	0.31s	38.23s

Tabela 10.1: Resultados comparativos no protótipo LIBRAS

10.1.4 Experimentos

Para testar o desempenho do AdapTree-E nós coletamos algumas imagens de 9 diferentes sinais alfabéticos e criamos um conjunto de treinamento no formato utilizado pelo WEKA (formato arff). Usando os utilitário de *benchmark* do WEKA, comparamos nosso algoritmo com os tradicionais C4.5, de indução de árvores de decisão, e o *backpropagation* para treinamento de redes neurais artificiais. A tabela 10.1 apresenta os resultados estatísticos obtidos utilizando 66% do conjunto de 300 exemplos para treinamento e o restante para teste. As estimativas sobre o percentual de exemplos corretamente classificados (PECC) e o tempo de treinamento são obtidas a partir de 10 execuções de cada um dos 3 algoritmos comparados, com os exemplos sendo "embaralhados" antes de cada ciclo de execução (um ciclo corresponde à execução de todos os algoritmos sobre o mesmo conjunto de treinamento e teste). A tabela 10.1 mostra também (linha dois) o desempenho dos algoritmos de aprendizagem quando as imagens de treinamento e teste são obtidas a partir de diferentes contextos (imagens coletadas em dias diferentes e sob diferentes condições de iluminação). O desempenho do AdapTree-E é comparável ao do C4.5 mas é bem inferior ao *backpropagation*. No entanto, este baixo desempenho é compensado por um tempo de treinamento inferior ao das outras estratégias, característica bastante importante em tarefas que exigem aprendizagem contínua.

10.1.5 Conclusão

Além de oferecer um exemplo de aplicação concreto de dispositivos adaptativos, o protótipo aqui apresentado pode servir de base para o desenvolvimento de aplicações mais complexas envolvendo aprendizagem de máquina, processamento digital de sinais e tecnologia adaptativa. Uma contribuição que não está diretamente relacionada aos objetivos desta tese, mas que merece ser registrada é a forma com que os parâmetros são extraídos, com base em momentos da imagem. Constatamos, através destes experimentos, que a estratégia de aprendizagem ainda carece de maior robustez

em relação a diferentes contextos, mas acreditamos que novos estudos que utilizem ainda mais profundamente o conceito de adaptatividade poderão vir a contribuir para a melhoria do desempenho do AdapTree-E. A implementação de um módulo para rastreamento das mãos, baseado possivelmente em modelos de cor de pele e diferença entre imagens subseqüentes (MARTIN; DEVIN; CROWLEY, 1998), é também uma meta importante para o futuro. Próximas implementações deste protótipo deverão estender o conjunto de sinais e iniciar o tratamento de gestos (que envolvem múltiplas imagens). Para isto, seria muito interessante a participação de especialistas na língua de sinais brasileira e de comunicadores surdos que possam auxiliar na construção de uma base de treinamento mais ampla e diversificada, nos moldes da base de treinamento criada em Perdue, para a língua americana de sinais (MARTíNEZ et al., 2002).

10.2 Interação Homem-Máquina através do Olhar

10.2.1 Introdução

O baixo custo dos dispositivos de captura de imagens, juntamente com o aumento significativo na capacidade de processamento dos computadores pessoais da atualidade, abrem espaço para o desenvolvimento de novos tipos de interface homem-máquina, utilizando técnicas de visão computacional. Uma alternativa que se tem mostrado bastante promissora (JACOB, 1995) consiste em detectar, através de uma ou mais câmeras filmadoras, a direção do olhar do usuário que se encontra em frente ao monitor. Esta informação pode ser utilizada tanto de maneira passiva, como, por exemplo, para registrar as imagens de uma página da Internet que mais chamam a atenção de determinados usuários, quanto de maneira ativa, quando as informações são utilizadas para controlar o que está sendo apresentado na tela (WARE; MIKAELIAN, 1987). Estudos indicam que objetos apresentados na tela de um computador podem ser apontados com maior velocidade e comodidade com os olhos do que com um mouse (JACOB, 1995).

Um benefício imediato da aplicação de técnicas de detecção da direção do olhar é a possibilidade de melhorar significativamente a capacidade de interação dos portadores de deficiências motoras, permitindo que até pessoas cujos movimentos se limitam ao globo ocular tenham a possibilidade de utilizar computadores. Sistemas completos que possibilitam este tipo de interface já estão disponíveis no mercado, no entanto, a maioria deles exige a utilização de dispositivos especiais, como capacetes, emissores e

detectores de sinais infravermelho (BALUJA; POMERLEAU, 1994) e câmeras de foco automático (*gaze-camera*) (WANG; SUNG, 2001).

No presente trabalho pudemos verificar, na prática, a viabilidade da utilização de algoritmos de detecção do olhar em um sistema composto de um computador pessoal comum (Pentium II 450Mhz, 128 Mb de Ram, 8Gb de HD) e uma câmera de baixo custo (WebCam Creative) posicionada acima do monitor (ver figura 10.5). Para isto, implementamos um jogo muito simples, o jogo da velha (*Tic-Tac-Toe*), que pode ser jogado utilizando apenas os olhos. A solução utilizada baseia-se em técnicas de visão computacional para localização e extração de atributos na região dos olhos e em técnicas de aprendizagem computacional para a determinação da direção do olhar. A aprendizagem é implementada, novamente, com o auxílio de um dispositivo adaptativo baseado em árvores de decisão, o qual além de apresentar taxas de acerto próximas ao das redes neurais com *backpropagation*, permite que erros de classificação possam ser reparados dinamicamente com base em instâncias adicionais de treinamento.

Figura 10.5: Ambiente em que os experimentos com o vTTT foram realizados

O desenvolvimento deste protótipo também foi feito utilizando-se o pacote integrado descrito no capítulo 9, que possibilitou uma redução significativa no tempo de implementação. O restante deste capítulo está organizado da seguinte forma: a próxima seção apresenta algumas estratégias utilizadas atualmente para detecção da direção do olhar. O desenvolvimento do protótipo e alguns experimentos com ele realizados são apresentados nas seções 10.2.3 e 10.2.4. Alguns dos resultados obtidos, bem como algumas propostas para prosseguimento desse trabalho podem ser encontrados na última seção.

10.2.2 Técnicas de Detecção da Direção do Olhar

Os primeiros sistemas a permitirem a interação homem-máquina através de movimentos da face ou dos olhos foram caracterizados por serem ou *intrusivos* ou *expansivos* (WANG; SUNG, 2001). Sistemas *intrusivos* exigem que o usuário "vista" equipamentos especiais, como óculos ou capacetes, enquanto os *expansivos* utilizam dispositivos não-convencionais, como por exemplo, diodos emissores de raios infravermelhos, que facilitam a detecção das pupilas humanas (COLOMBO; BIMBO, 1997), e câmeras filmadoras especiais, com foco automático.

Recentemente, a queda nos custos de dispositivos de captura de imagens, como as populares *WebCams*, impulsionou a busca de sistemas não-intrusivos de detecção da direção do olhar. A direção do olhar pode ser estimada tanto pela observação dos movimentos da cabeça, como um todo, quanto pelo movimento do globo ocular e da íris (WANG; SUNG, 2001). Estas estimativas podem ser obtidas por técnicas algébricas de inferência do foco do olhar a partir de características especificas da face projetada em 2D (GEE; CIPOLLA, 1994; WANG; SUNG, 2001), ou por aprendizagem computacional (BALUJA; POMERLEAU, 1994; STIEFELHAGEN; YANG; WAIBEL, 1997).

Stiefelhagen (STIEFELHAGEN; YANG; WAIBEL, 1997) relata ter obtido uma precisão entre 1.3 e 1.9 graus em um sistema que permite o controle do ponteiro do mouse através do olhar. Em seus experimentos foram utilizadas 4000 imagens de pessoas acompanhando um cursor que se deslocava sobre a tela de um monitor. Recortes destas imagens contendo apenas a região da face envolvendo os olhos, o nariz e a boca, junto com a posição do cursor na tela, alimentaram uma rede neural artificial do tipo *feed-forward* com três níveis, utilizando a técnica de *backpropagation* para treinamento. Embora precisões de até 0.75 graus possam ser obtidas usando outros métodos (BALUJA; POMERLEAU, 1994), as técnicas que utilizam redes neurais, além de não dependerem de dispositivos especiais, podem ser utilizadas em diversos tipos de ambiente e iluminação, o que em geral não ocorre com as outras técnicas. O problema com o enfoque baseado em redes neurais "tradicionais" é que o aprendizado não é incremental, o que dificulta a correção de erros, no modelo aprendido, durante a fase de utilização do sistema.

Seguindo também a linha não-intrusiva, Gee e Cipolla (GEE; CIPOLLA, 1994), descrevem um sistema que infere a direção do olhar a partir da reconstituição de um modelo simplificado da cabeça humana, a partir da imagem projetada. Neste modelo,

busca-se valorizar a posição relativa dos olhos, boca e nariz, que, segundo os autores, são atributos que variam menos em relação a diferentes sujeitos do que, por exemplo, as orelhas, que podem estar cobertas, ou semicobertas, por cabelos.

Wang e Sung (WANG; SUNG, 2001) modelam a direção do olhar como sendo a direção da normal ao plano de suporte de cada íris. Aproximando a forma da íris através de circunferências e assumindo raios, distâncias focais e fatores de escala conhecidos, é possível estimar a normal em questão a partir da projeção, elíptica, destes dois círculos. A manipulação algébrica deste problema oferece, no entanto, duas soluções para cada íris. Wang e Sung propõe a escolha das normais mais próximas entre si para resolver este impasse, e apresentam diversos resultados empíricos que sustentam a utilidade desta heurística. A detecção das duas elipses que correspondem ao contorno da íris na imagem projetada não é uma tarefa trivial, principalmente se for executada em tempo real. Tanto a estratégia de Wang e Sung, quanto a de Gee e Cipolla, exigem uma alta capacidade de processamento, o que dificulta a construção de soluções de baixo custo.

Transformadas de Hough são ferramentas poderosas na detecção de linhas e círculos a partir de imagens cujas bordas foram previamente realçadas (MCLAUGHLIN, 1998). De modo geral, as transformadas trabalham em um domínio definido pelos possíveis parâmetros da equação que descreve o ente geométrico em questão. No caso de retas, descritas pela equação $y = ax + b$, temos um domínio bidimensional, com dois eixos ortogonais, para os parâmetros a e b. Cada ponto neste novo "espaço" determina uma possível reta e a existência desta reta na imagem original é calculada a partir de uma varredura nos pontos que participam das bordas detectadas. Para cada um destes pontos, o algoritmo de detecção deve varrer uma das dimensões do espaço de Hough e calcular o valor do outro parâmetro, incrementando um acumulador associado a cada ponto deste espaço. Ao final, os pontos no espaço de Hough com maiores valores em seus acumuladores são os que representam as retas mais prováveis, na imagem original.

Teoricamente, as transformadas de Hough podem ser utilizadas na detecção de qualquer figura geométrica definida por equações paramétricas. No entanto, seu cálculo torna-se impraticável quando as equações são não-lineares e envolvem mais de 3 parâmetros, como é o caso das elipses. McLaughlin (MCLAUGHLIN, 1998) propõe uma melhoria para o algoritmo original, melhoria esta que consiste em utilizar um sis-

tema de 3 equações lineares na descrição de elipses, e uma varredura, por amostragem informada, sobre os pontos que constituem as bordas da imagem original. Esta nova técnica denomina-se Transformada de Hough Aleatorizada (*Randomized Hough Transform*). Nossos experimentos indicaram, no entanto, que mesmo esta versão otimizada não pode ser executada em tempo real em equipamentos comuns. Outras técnicas para detecção de elipses envolvem aproximações pelo método dos mínimos quadrados (PILU; FISHER, 1996) e filtros de Kalman (PORILL, 1990).

10.2.3 Desenvolvimento

Para realizar alguns dos nossos experimentos com interfaces baseadas na direção do olhar, implementamos um simples jogo da velha. Existem apenas nove regiões de interesse para o usuário neste jogo: as posições do tabuleiro onde podemos marcar os círculos ou as cruzes. Fazendo com que a apresentação visual do tabuleiro ocupe grande parte do monitor, conseguimos relaxar o grau de precisão na detecção da direção do olhar. Além disto, é bastante natural a transposição da interface por mouse para a interface pelo olhar.

Inicialmente, existe um período de aprendizagem, em que usuário deve olhar para cada uma das nove regiões do tabuleiro. A captura de imagens de treinamento é iniciada e finalizada clicando-se o mouse. Depois que uma quantidade pré-definida de exemplos é capturada, o sistema infere uma primeira árvore de decisão, e o usuário pode começar a jogar “com o olhar”. Uma jogada é identificada quando o usuário mantém o olhar em uma região do tabuleiro por alguns segundos. Caso o sistema não indique a região correta, o usuário pode acionar o módulo de aprendizagem, clicando duas vezes na região apropriada, enquanto olha. Como no modo de treinamento inicial, o primeiro clique inicia a captura de exemplos, enquanto o segundo a termina. Isso ilustra uma das vantagens de um algoritmo incremental de aprendizagem, como o AdapTree-E, que permite que o modelo inferido (no caso, uma árvore de decisão), possa ser dinamicamente modificado em parte, e praticamente em tempo real.

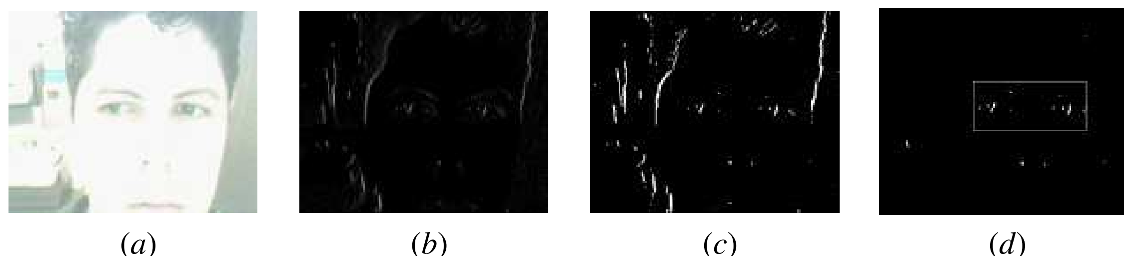
Por questões de eficiência, o módulo de aprendizagem não trabalha diretamente sobre a imagem obtida, mas sobre um conjunto de parâmetros dela extraídos pelo módulo de processamento digital de imagens. A extração de atributos pode ser dividida nas três fases descritas nas próximas seções: pré-processamento da imagem, delimitação da região dos olhos e cálculo de atributos.

-1	0	1
-1	0	1
-1	0	1

Tabela 10.2: Matriz de convolução para detecção de bordas

10.2.3.1 Pré-Processamento

A figura 10.6 (a) mostra a imagem sem nenhum processamento, tal como é extraída por meio de uma câmera WebCam Creative, posicionada acima do monitor. O contraste e o brilho da imagem obtida pela câmera são ajustados apropriadamente para facilitar o pré-processamento: pouco contraste e alto brilho causam um efeito de suavização na imagem. O primeiro passo consiste em transformar a imagem de colorida, RGB, para tons de cinza, seguindo a fórmula $Cinza = (Red * 0.299 + Green * 0.587 + Blue * 0.114)$. Aplicamos em seguida um detector de bordas verticais, cujo núcleo é mostrado na tabela 10.2. O resultado da aplicação deste filtro é apresentado na figura 10.6 (b). Finalmente, binarizamos a imagem aplicando a mesma técnica de limiarização iterativa citada no capítulo anterior. A imagem resultante é mostrada na figura 10.6 (c).



(a) (b) (c) (d)

Figura 10.6: Detecção da região dos olhos no vTTT
(a) Original Suavizada (b) Bordas Verticais (c) Binarização (d) Região dos Olhos

10.2.3.2 Delimitação da Região dos Olhos

Na detecção da região dos olhos busca-se marcar, na imagem previamente processada, uma região retangular, contendo apenas os olhos humanos, como mostra a figura 10.6 (d). O algoritmo responsável por esta tarefa baseia-se em 4 heurísticas: (1) existe uma borda vertical facilmente detectável entre íris e esclera (parte branca do olho) (WANG; SUNG, 2001), (2) as bordas existentes nas regiões imediatamente superiores (testa) e inferiores (maçãs da face) são bem mais tênues que as dos olhos e somem quase que completamente durante o pré-processamento, (3) a distância entre

os dois olhos, quando vistos de frente, pode ser limitada inferior e superiormente e (4) os olhos mantêm-se razoavelmente alinhados em relação ao eixo horizontal.

Embora seja possível encontrar, facilmente, casos em que as heurísticas acima possam ser invalidadas (e.g. franjas sobre a testa, cabeça inclinada), nos testes efetuados com três diferentes pessoas, os resultados obtidos foram encorajadores. Um fator importante dessas heurísticas, com exceção da primeira, é que usuário pode, na maioria das vezes, adequar-se facilmente às restrições por elas impostas, mudando, por exemplo, o penteado do cabelo, afastando-se ou aproximando-se da câmera e não inclinando demasiadamente a cabeça para os lados.

A implementação dessas heurísticas ocorre da seguinte forma: primeiramente, utilizamos a heurística (2) para retirar diversas bordas verticais que aparecem no fundo da imagem e no contorno da face. Essa operação é efetuada através da aplicação de um operador morfológico, que varre a imagem, limpando os pontos que não possuem uma região relativamente vazia abaixo ou acima de si, e seu resultado é ilustrado na figura 10.6 (*d*). Nos experimentos realizados, os valores exatos do tamanho, posição e quantidade mínima de pontos destas regiões foram determinados empiricamente. Na fase seguinte, testamos dois a dois os pontos que restaram e escolhemos os dois primeiros que satisfazem as heurísticas (3) e (4), utilizando para isto mais algumas constantes que foram determinadas através de medições no conjunto de imagens de treinamento (distâncias e inclinações máximas e mínimas).

10.2.3.3 Cálculo de Atributos

O cálculo de atributos é feito apenas para a região dos olhos, utilizando um método similar àquele apresentado na seção anterior, sobre LIBRAS. A região dos olhos é dividida através de um reticulado 2×2. Para cada uma destas sub-regiões, calculamos o total de *pixels* (massa) e o centro de massa (média dos valores x, y de cada *pixel* na região). Somando-se a esses 12 valores: a massa, o centro de massa e a variância em relação ao centro de massa globais, temos 17 atributos que são efetivamente utilizados pelo módulo de aprendizagem, além, é claro, de um valor indicando a posição do tabuleiro que está sendo focada pelo usuário no momento da captura da imagem.

10.2.4 Experimentos

Realizamos três tipos de experimentos com o protótipo criado. No primeiro experimento, estimamos a taxa de acerto do módulo de delimitação da região dos olhos, utilizando 3 indivíduos diferentes. Enquanto os indivíduos olhavam para diversas regiões da tela, imagens com as regiões dos olhos delimitadas (como na figura 10.6 (*d*)) eram mostradas na tela e gravadas em disco. Observando essas imagens, é possível indicar, por inspeção, em quais delas a região dos olhos foi corretamente delimitada. A taxa de acerto obtida gira em torno de 90%, sendo que a retro-alimentação, oferecida pelas imagens delimitadas, pode ser utilizada para aumentar esta taxa. Assim, observando as situações em que o sistema erra, o usuário pode ajustar sua posição diante do monitor, para "facilitar" o trabalho do módulo de detecção da região dos olhos.

No segundo tipo de experimento, o usuário simplesmente utiliza o protótipo para treinar e jogar. Os primeiros resultados indicam que precisamos no mínimo de 250 exemplos de treinamento para obtermos uma taxa de acerto razoável durante o jogo. Embora a obtenção dos exemplos seja uma tarefa bastante simples para o usuário (olhar e clicar), constatamos que o sistema ainda é muito dependente da forma exata em que estes exemplos são extraídos. Por exemplo, se o usuário treina o sistema de manhã e vai jogar à noite, a taxa de acerto cai significativamente, e o sistema precisa ser retreinado.

Para verificar de forma menos subjetiva o desempenho do algoritmo de aprendizagem, coletamos imagens de algumas seções de utilização do software e criamos arquivos de treinamento no formato utilizado pelo Weka. Comparamos o AdapTree-E com os algoritmos C4.5, Backpropagation. Os resultados são apresentados na tabela 10.3, para arquivos de treinamento com 50 e 250 exemplos e utilizando o método de comparação *RandomSplit*, do Weka, com 10 repetições e um corte de 66%. A tabela mostra também a taxa de acerto quando os arquivos de teste e de treinamento são extraídos em contextos diferentes. É importante ressaltar, novamente, que o AdapTree-E é o único, dentre os três comparados, capaz de absorver novas instâncias de treinamento de forma eficiente, sem que haja necessidade de efetuar novo treinamento.

10.2.5 Conclusões

Neste trabalho foram apresentadas algumas táticas de solução para o problema da detecção da direção do olhar a partir de imagens da face. Foi apresentado também

	AdapTree-E	C4.5	Backpropagation
Taxa de Acerto (50)	92.17%	91.64%	100%
Taxa de Acerto (250)	95.18%	95.18%	100%
Taxa de Acerto (Diferentes Contextos)	68%	79.33%	90.7%
Tempo Médio Aprendizagem	0.36s	0.33s	42.23s

Tabela 10.3: Resultados comparativos no vTTT

um protótipo de um sistema que permite a interface homem-máquina utilizando imagens capturadas por uma câmera comum, posicionada sobre a tela de um monitor e capturando imagens frontais do usuário. Podemos constatar através deste protótipo que é possível acompanhar a região dos olhos em tempo real, utilizando equipamentos comuns de processamento digital.

As heurísticas utilizadas para detecção da região dos olhos se baseiam em fatores que apresentam baixo índice de variação entre faces de diferentes pessoas, como é o caso do forte contorno existente entre íris e a esclera, a distância média entre as duas pupilas e a relativa suavidade das regiões superior e inferior aos olhos (testa e maçãs da face). Para determinar a região dos olhos utilizando estas heurísticas, foram empregados filtros de detecção de borda vertical, limiarização e operadores morfológicos. O algoritmo baseia-se na hipótese de que o usuário se apresenta em uma posição predefinida em relação ao monitor e à câmera (entre 35cm e 55cm). Essa posição pode ser obtida facilmente observando o erro na detecção da região dos olhos, que é apresentado em tempo real para usuário, o qual pode então realizar ajustes em sua posição física, até encontrar a região correta (quando os erros atingem um patamar aceitável).

Os primeiros experimentos com a utilização de transformadas de Hough para a detecção da projeção elíptica da íris indicaram que seria muito difícil obter os resultados indicados por Wang. Parece, a princípio, que os bons resultados obtidos por Wang devem-se muito à utilização de uma câmera de foco automático, centrada na íris e em *zoom*. Por isso, optamos por realizar experimentos com aprendizagem de máquina, como sugerido por Stiefelhagen (STIEFELHAGEN; YANG; WAIBEL, 1997) e Baluja (BALUJA; POMERLEAU, 1994), que relatam terem obtido boa precisão na detecção do ponto focado, utilizando apenas uma câmera. A diferença de nossa técnica reside no mecanismo incremental de aprendizagem, baseado na tecnologia adaptativa, o que permite ao usuário corrigir eventuais erros constatados, mesmo após a fase de treinamento inicial.

Atualmente, a detecção da região dos olhos não reaproveita qualquer informação de processamentos anteriores. Poderíamos, por exemplo, diminuir o espaço de varredura em busca da posição atual dos olhos armazenando conhecimento sobre a posição anterior. Estudar o impacto e viabilidade deste tipo de estratégia nos parece ser uma interessante meta de pesquisa para os trabalhos seguintes.

Outros pontos, que podem ser explorados posteriormente, e que podem vir a melhorar bastante a eficácia da detecção da região dos olhos incluem a utilização de algoritmos de detecção de regiões de pele humana (GOMEZ M. SANCHEZ, 2002) e a introdução de uma fase de calibração, em que características específicas do usuário possam ser identificadas. Em relação à fase de aprendizagem, seria interessante explorar novos atributos e modelos baseados em técnicas sintáticas de reconhecimento. Além disso, pode-se iniciar a seguir um processo de sofisticação dos algoritmos para que sejam aplicados em problemas mais complexos e que exijam maior precisão, como por exemplo, um jogo de damas. No jogo de damas, poderemos começar a explorar alternativas para os movimentos de arraste e clique do mouse, utilizando técnicas estudadas na área de Interação Homem-Máquina (IHM).

Embora muito ainda possa ser feito para melhorar o protótipo aqui apresentando, acreditamos que estes primeiros experimentos ajudaram a mostrar a viabilidade de novas interfaces baseadas na detecção da direção do olhar. É importante ressaltar que todo o projeto está sendo desenvolvido em plataformas de baixo custo, e utilizando software de programas-fonte abertos, o que acreditamos poder resultar em soluções mais acessíveis do que as disponíveis atualmente no mercado.

11 COMPILADORES E FERRAMENTAS EDUCACIONAIS

Mostramos neste capítulo algumas aplicações criadas com o auxílio da ferramenta AdapTools, descrita anteriormente, que se aplicam na área de construção de compiladores e na de tradução texto-voz. Essas aplicações têm também finalidade pedagógica, podendo ser utilizadas como ferramentas de apoio educacional em disciplinas da área da computação.

11.1 Meta-reconhecedor Wirth para AdapTools

Aprender a projetar e implementar um compilador é uma tarefa desafiadora e árdua enfrentada por alunos de cursos de graduação na área da computação. A compreensão profunda dos conceitos e técnicas envolvidos na construção de compiladores é essencial na formação de profissionais de computação. Além da importância central dos compiladores no desenvolvimento de sistemas computacionais, os conceitos e técnicas desta área da computação podem ser aplicados, direta ou indiretamente, a praticamente todas as outras áreas. Portanto, é fundamental que se busque sempre o aprimoramento das ferramentas e métodos educacionais, para garantir o melhor aproveitamento possível do tempo, em geral alguns meses, em que o aprendiz se envolve diretamente com o problema, nas disciplinas diretamente relacionadas com construção de compiladores.

Um exemplo ilustrativo, que facilita a assimilação dos conceitos envolvidos na construção de compiladores, é descrito em (NETO; PARIENTE; LEONARDI, 1999). Seguindo a proposta descrita nesse trabalho, implementamos através do AdapTools um meta-reconhecedor que recebe, como entrada, uma especificação de uma linguagem

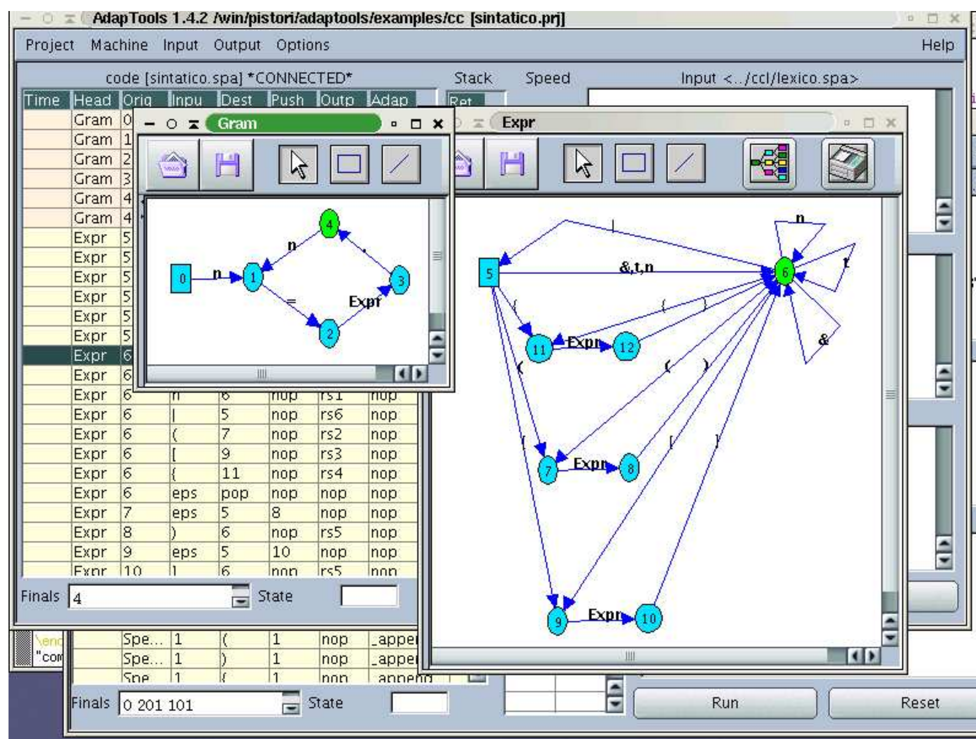


Figura 11.1: Meta-reconhecedor Wirth para AdapTools

livre de contexto, em notação de Wirth, e produz um reconhecedor para tal linguagem. Esse reconhecedor é compatível com a máquina virtual do AdapTools, podendo, portanto, ser diretamente executado após ter sido gerado. Com isso, em um único ambiente gráfico, o aluno pode realizar experimentos no nível do meta-reconhecedor (alterações, reconstruções), do compilador (acréscimo de melhorias, inserção de rotinas semânticas) e do código gerado pelo compilador.

A implementação desse meta-reconhecedor (figura 11.1) ilustra também um outro recurso poderoso do AdapTools, que é a execução concorrente de múltiplas máquinas. A análise sintática e léxica, nesse meta-reconhecedor, é realizada por dois autômatos distintos (mais especificamente, autômatos de pilha estruturados), que podem ser executados concorrentemente. Isso possibilita ao aprendiz uma visão integrada do relacionamento entre analisadores léxicos e sintáticos. Esses dois autômatos, através de um exemplo ilustrativo, bem como um exemplo de entrada e saída para o meta-reconhecedor, são apresentados no anexo E.

11.2 Recuperação Automática de Erros

O objetivo de um compilador é converter códigos escritos em uma linguagem fonte L_F para uma linguagem objeto L_O. No entanto, como o código-fonte é geralmente produzido por seres humanos, é natural que ocorram erros. Um compilador pode reagir a um erro utilizando diversas estratégias, entre as quais são muito freqüentes as seguintes: (1) interrompendo sua execução assim que um erro é detectado, (2) buscando alguma forma de ajustar suas estruturas internas para que seja possível continuar o processo de compilação, a partir do ponto do código-fonte em que o erro foi detectado ou (3) corrigindo os erros contidos no código-fonte (GRUNE; JACOBS, 1990).

Na estratégia 1, que apenas detecta o erro, o usuário é obrigado a intervir, modificando o código-fonte e reiniciando o processo de compilação, a cada erro encontrado. Embora para o projetista de compiladores esta seja a solução mais simples, certamente, para o usuário, ela não é nada maleável. A estratégia 3, de correção do código-fonte, seria a ideal, pois permitiria que o código objeto fosse gerado mesmo na presença de erros. Embora seja muito complicado, no caso geral, prever exatamente qual seria o código-fonte correto imaginado pelo usuário, técnicas estatísticas e de inteligência artificial têm sido exploradas, em conjunto com técnicas tradicionais, na busca desse objetivo.

A estratégia 2, de recuperação automática de erros, é ainda a mais estudada e consolidada no campo da construção de compiladores (GRUNE; JACOBS, 1990). Entre as diversas técnicas existentes de recuperação de erros podemos destacar o modo-pânico (GRUNE; JACOBS, 1990), recuperação por avanço (*forward repair*) (MAUNEY; FISHER, 1982), recuperação local (SIPPU; SOISALON-SOININEN, 1983), recuperação global com custo mínimo (*global least cost repair*) (AHO; PETERSON, 1972) e recuperação regional com custo mínimo (*regional least-cost error repair*) (VILARES; DARRIBA; RIBADAS, 2001).

Nas próximas seções apresentaremos a implementação adaptativa de uma técnica de recuperação de erros para autômatos de estados finitos que pode ser facilmente estendida para autômatos de pilha estruturados. Esta técnica é baseada em tecnologia adaptativa, e oferece um opção elegante e eficiente para o problema da recuperação de erros. Na próxima seção faremos uma breve revisão de uma técnica clássica para recuperação de erros simples em autômatos de estados finitos. Uma solução adaptativa,

baseada em uma reformulação desta técnica, é apresentada na seção 11.2.2. A seção seguinte discute a implementação desta técnica utilizando o AdapTools. Conclusões e possíveis trabalhos futuros nessa direção são apresentados na seção 11.2.4.

11.2.1 Recuperação de Erros Simples - Método Clássico

Este método clássico permite que um autômato de estados finitos continue consumindo símbolos da cadeia de entrada mesmo que um erro seja detectado. A detecção do erro, neste contexto, é trivial, e corresponde à leitura de um símbolo para o qual não exista uma transição cujo estado de origem seja o estado corrente. Já a recuperação do erro envolve a aceitação de uma premissa simplificadora sobre os tipos de erro que podem ocorrer na cadeia de entrada: omissão, inserção e a substituição de um único símbolo de cada vez (o método não trata, por exemplo, duas inserções incorretas consecutivas). Interessantemente, esta premissa não chega a ser tão restritiva, uma vez que grande parte dos erros cometidos na digitação de um código-fonte são simples omissões, inserções e substituições (os chamados "erros simples").

Um autômato de estados finitos pode se recuperar de erros de omissão, inserção e substituição efetuando, respectivamente, a inserção do símbolo omitido, a remoção do símbolo inserido e a substituição de um símbolo incorreto pelo correto. Todas estas operações podem ser obtidas através da inserção estratégica e metódica de novas transições e estados ao autômato de estados finitos original. Para que tais transições sejam utilizadas apenas quando da ocorrência de erros, define-se um tipo especial de transição, *transição de erro*, que é ativada apenas em caso de erro (quando não existe uma transição de saída para o estado corrente consumindo o próximo símbolo a ser lido da cadeia de entrada). Uma alternativa para a criação deste tipo especial de transição seria a utilização de funções-falha (*failure functions*), muito utilizadas em problemas de casamento de cadeias (*string matching*) (MOHRI, 1997; KIDA et al., 1998; HIRSCHBERG; LARMORE, 1987).

Definimos abaixo alguns conceitos que são utilizados na apresentação do algoritmo 11.1, de recuperação de erros simples.

Definição Seja $M = (Q, \Sigma, Q_0, F, \delta)$ um AEF. $first : Q \to 2^Q$ é uma função sobre o conjunto de estados na qual $q' \in first(q)$ se e somente se $\exists \sigma \in \Sigma \mid (q, \sigma x) \vdash_M^*$

$(q', x), x \in \Sigma^*$ [1]. Ou seja, $first$ fornece todos os estados alcançáveis a partir de um estado qualquer e com a leitura de um símbolo da cadeia de entrada.

Definição Seja $first(q) = \{q_0, q_1, ..., q_n\}$, em que $q \in Q$ é um estado qualquer de M. A função $second : Q \to 2^Q$ é definida para cada elemento $q \in Q$ da seguinte forma:

$$second(q) = \bigcup_{0 \leq i \leq n} first(q_i) \quad (11.1)$$

A função $second$ fornece o conjunto de estados que podem ser atingidos a partir de um estado qualquer consumindo-se dois símbolos da cadeia de entrada. Finalmente, dado um estado $q \in Q$, utilizaremos o símbolo θ para denotar o conjunto $\Sigma - firstSymbols(q)$, no qual $firstSymbols : Q \to 2^\Sigma$ é um função tal que, para cada $q \in Q$, $\sigma \in firstSymbols(q)$ se e somente se $(q, \sigma x) \vdash_M^* (q', x), x \in \Sigma^*, q' \in first(q)$ ($firstSymbols$ fornece o conjunto de símbolos que podem ser lidos a partir de um estado qualquer).

Algoritmo 11.1 Recuperação de erro simples em AEF

entrada: AEF $M = (Q, \Sigma, q_0, F, \delta)$
saída: M com recuperação de erro

```
1: para cada q ∈ Q faça
2:     Q ← Q ∪ {e1, e2} {Adiciona dois novos estados}
3:     Adicione uma transição de erro de q para e1
4:     para cada c ∈ Q − {first(q) ∪ second(q)} faça
5:         δ ← δ ∪ (e1, c, e2)
6:     fim para
7:     para cada qs ∈ second(q) faça
8:         Seja b ∈ Σ o símbolo que garante a presença de qs em second(q)
9:         δ ← δ ∪ (e1, b, qs)
10:        δ ← δ ∪ (e2, b, qs)
11:    fim para
12:    para cada qf ∈ first(q) faça
13:        Seja a ∈ Σ o símbolo que garante a presença de qf em first(q)
14:        δ ← δ ∪ (e2, a, qf)
15:    fim para
16:    se q é um estado final então
17:        Faça de e2 um estado final
18:    fim se
19: fim para
```

A figura 11.2 ilustra a execução do algoritmo 11.1 em um autômato com um alfabeto de três símbolos, $\Sigma = \{a, b, c\}$. Na figura, mostramos apenas as transições inse-

[1] $\vdash^*$ denota a fechamento transitivo reflexivo da relação de passo do autômato, $\vdash$

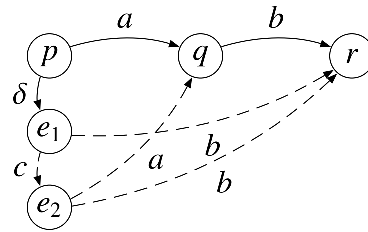


Figura 11.2: Exemplo da execução do algoritmo de recuperação de erros

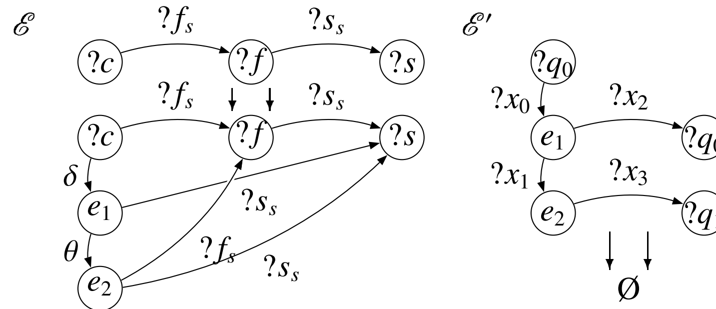


Figura 11.3: Funções adaptativas para recuperação de erro

ridas (linhas pontilhadas) para tratar erros ocorridos no estado p. A transição de erro está sendo indicada pelo símbolo δ. O autômato completo resultante teria três vezes mais estados que o original (dois novos estados de erro para cada estado original), além de uma grande quantidade de novas transições.

11.2.2 Recuperação de Erros Adaptativa

A recuperação de erros utilizando autômatos de estados finitos adaptativos é obtida através da adição de transições de erro a cada um dos estados do autômato subjacente. Cada transição de erro é ligada a uma ação adaptativa, $\mathscr{E}$, responsável por modificar o autômato para que este possa recuperar-se de eventuais erros simples. As modificações definidas por $\mathscr{E}$ implementam basicamente o algoritmo clássico citado na seção anterior. Algumas das transições criadas por $\mathscr{E}$ carregam uma segunda ação adaptativa, $\mathscr{E}'$, responsável por remover as transições recém-criadas, depois que o erro for recuperado. A figura 11.3 mostra graficamente as funções adaptativas $\mathscr{E}$ e $\mathscr{E}'$.

As diferenças em relação à solução clássica, embora sutis, abrem um novo espaço para investigações na área de recuperação de erros, ao permitir uma nova modelagem para o problema. A recuperação de erros passa a ser um elemento intrínseco do dispositivo, que pode ser elegantemente visualizado como algo capaz de se auto-modificar para tolerar a ocorrência de erros simples. Em termos práticos, existe também a

questão da economia de espaço, pois, na solução adaptativa, os novos estados e transições são criados e mantidos apenas pelo intervalo de tempo estritamente necessário.

11.2.3 Implementação no AdapTools

Para experimentar a estratégia aqui apresentada, criamos manualmente alguns autômatos adaptativos com recuperação de erros. O código de um desses exemplos é listado no anexo F. Também implementamos uma outra versão para o meta-reconhecedor da seção 11.1, que produz automaticamente autômatos capazes de se recuperarem de erros simples. Para implementar esta versão do meta-reconhecedor foi preciso apenas alterar algumas rotinas de geração de código, para que elas passassem a acrescentar, ao final do código objeto gerado, a descrição das funções adaptativas para tratamento de erros, e as transições de erro carregando as ações adaptativas adequadas.

11.2.4 Conclusões

Um novo enfoque para a recuperação de erros utilizando tecnologia adaptativa foi apresentada. Esse enfoque tem, como principal característica, a capacidade de possibilitar que a especificação do módulo de recuperação de erros de um dispositivo seja feita através do mesmo formalismo utilizado na especificação dos outros módulos. Estratégias mais sofisticadas para a recuperação e até mesmo para a correção de alguns erros, utilizando a tecnologia adaptativa, deverão ser investigadas a seguir. Uma das vertentes promissoras de pesquisa nesta área está relacionada com a utilização da capacidade de aprendizagem dos dispositivos adaptativos para a construção de recuperadores de erro que possam, através da interação com o usuário ou da indução por exemplos, assimilar padrões de comportamento em relação à produção de erros. Tais padrões poderiam ser utilizados, por exemplo, para definirem transições relacionadas à recuperação de erros a ser definitivamente incorporada ao autômato (buscando assim um equilíbrio entre os custos de execução em relação ao tempo e ao espaço).

Outra linha interessante é a aplicação da recuperação de erros em sistemas de busca aproximada. Um conceito central nesta área é o da medida de distância entre duas cadeias. Em geral, um limite mínimo de proximidade, definido em função dessa distância, é utilizado para determinar quais cadeias devem ser retornadas pela busca. Um exemplo de como tal distância pode ser definida é a *distância de edição* (MOHRI,

2002). Esta métrica baseia-se na quantidade de erros simples (no sentido utilizado neste capítulo) detectados por um autômato que reconhece exatamente uma cadeia x, ao receber uma cadeia y (possivelmente diferente de x). Poderíamos, usando tecnologia adaptativa, generalizar tal métrica, modificando-a para algo como: a quantidade de ações adaptativas elementares necessárias para transformar um autômato que reconhece x em um autômato que reconhece y. Com isto obtemos uma nova maneira para comparar e construir novas métricas, a partir do projeto de novas funções adaptativas.

11.3 Tradução Texto-Voz

Um outro exemplo, também contido no pacote AdapTools, é um protótipo de um conversor texto-voz utilizando autômatos de estados finitos. Este protótipo, embora bastante simples e sem recursos sofisticados de geração automática de entonação ou de ligação suave entre fonemas, já se apresenta como um núcleo importante sobre o qual poderão ser desenvolvidas soluções mais sofisticadas. No campo pedagógico, ele mostra, de uma maneira bastante atraente, que as técnicas de compilação não se restringem ao domínio das linguagens de programação. No caso específico, temos a tradução (mesmo que ainda pouco sofisticada) de um código-fonte, escrito em linguagem natural, para uma seqüência de sinais sonoros reproduzidos por alto-falantes.

A figura 11.4 mostra algumas das transições do AEF que traduz texto em voz. Nas legendas da forma x/y, x indica o símbolo de entrada, e y, o de saída. A semântica deste dispositivo foi implementada de forma tal que cada símbolo de saída y seja mapeado em um arquivo de som, no formato *wave* (.wav), que é automaticamente enviado para a saída de som do computador assim que a transição gera y é executada. As transições $(4, a, 5)$ e $(3, a, 5)$, por exemplo, produzirão ambas o mesmo som (aquele do único fonema da palavra *chá*). O código-fonte que implementa essa semântica é listado no anexo G.

Uma das dificuldades na tradução texto-voz é que uma mesma sílaba, quando encontrada em palavras ou sentenças diferentes, pode ter diferentes pronúncias. Por exemplo, a sílaba *xa*, nas palavras $fixa$ e *Xuxa*, corresponde a distintos fonemas. Este problema está intimamente ligado à dependência de contexto, o que sugere a utilização de autômatos adaptativos, no lugar de autômatos de estados finitos, na produção de sistemas mais poderosos desta natureza.

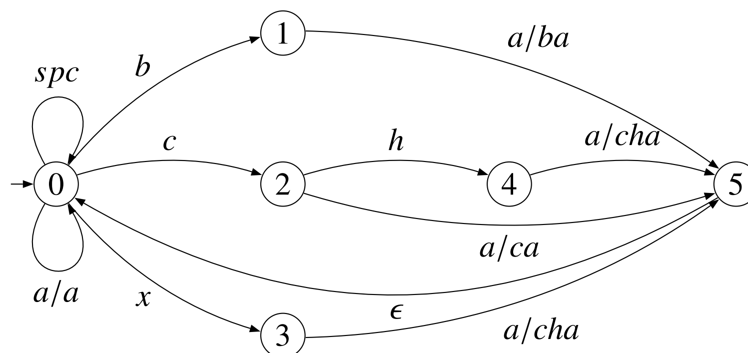


Figura 11.4: Algumas transições do autômato de tradução texto-voz

12 INTELIGÊNCIA ARTIFICIAL

As aplicações deste capítulo, que ainda estão em um estágio inicial de desenvolvimento, ilustram algumas possibilidades de utilização da tecnologia adaptativa em áreas geralmente estudadas em inteligência artificial, como é o caso do diagnóstico médico apoiado por computador.

12.1 Diagnóstico Médico

A utilização de técnicas de inteligência artificial para a criação de ferramentas de diagnóstico médico autônomas atingiu o seu auge na década de 80, quando programas como o *MYCIN* demonstravam poder oferecer diagnósticos mais precisos para determinadas doenças do que seres humanos especialistas (BUCHANAN; SHORTLIFFE, 1984). No entanto, o sucesso em laboratório destes programas não se traduziu em sucesso junto à comunidade médica em geral, e durante muitos anos, as pesquisas nessa área foram relegadas a um segundo plano. Uma conclusão interessante relatada em (HEDBERG, 1998), sobre os motivos deste fracasso, é a de que os médicos não estariam interessados em ferramentas de diagnóstico automático. O ressurgimento das pesquisas nesta área se deu quando o foco passou a ser a construção de sistemas de apoio ao trabalho médico, capazes de oferecer conselhos e realizar descobertas de padrões interessantes em bases de dados gigantes contendo informações médicas.

Aproveitando a existência de um núcleo de desenvolvimento de instrumentos médicos que mantém contatos com o grupo de linguagens e tecnologia adaptativa, iniciamos o desenvolvimento de um sistema para apoio ao diagnóstico de incontinência urinária de esforço (IUE). Neste sistema, o AdapTree-E será utilizado na indução de árvores de decisão com base em exemplos de diagnósticos reais, previamente arma-

zenados, através do sistema UROSYSTEM DS-5600 [1]. O UROSYSTEM integra um software de gerenciamento de pacientes, contendo informações clínicas, com um hardware utilizado na realização de exames urodinâmicos. Um médico especialista está desenvolvendo manualmente um modelo para diagnóstico de incontinência urinário de esforço, que deverá ser comparado com aquele gerado automaticamente, para podermos obter informações sobre o desempenho da nossa técnica nesta área de aplicação. Pretendemos também analisar maneiras de combinar os modelos induzidos mecanicamente com os modelos construídos manualmente, para produzir os modelos finais, que poderão vir a ser integrados ao UROSYSTEM, num módulo de apoio ao diagnóstico médico. Na próxima seção descrevemos um módulo desse sistema que já foi implementado. Infelizmente, até o momento ainda não tivemos acesso aos dados necessários para execução do AdapTree-E (uma base com cerca de 4000 casos de diagnósticos já realizados, mas que estão armazenados em um formato interno do UROSYSTEM).

12.2 Ambiente Interativo para Manipulação de Árvores de Decisão

O xILE, *Incremental Learning Environment for X*, é uma ferramenta que permite a visualização gráfica de algoritmos incrementais de indução de árvores de decisão. Embora tenha sido criado inicialmente para permitir a visualização de um árvore de decisão adaptativa, sua estrutura modular permite que a execução de outros algoritmos de indução de árvores de decisão, como o Id3 por exemplo, possa também ser acompanhada graficamente. Uma das finalidades desta ferramenta é o de oferecer apoio pedagógico a disciplinas da área de inteligência artificial, permitindo ao aluno a realização de experimentos com árvores de decisão aplicadas a diferentes domínios. O aluno pode, inclusive, com certa facilidade, substituir os mecanismos de indução de árvores de decisão presentes no xILE, por outros, que ele mesmo pode implementar. A tela principal do xILE é mostrada na figura 12.1

Uma outra finalidade do xILE é permitir que especialistas de um determinado domínio (e.g. um médico), para o qual um solução está sendo desenvolvida usando árvores de decisão adaptativas, possam interagir com o modelo produzido, inserindo novos exemplos e modificando a árvore resultante. O xILE oferece, por exemplo,

[1] http://www.viotti.com.br/paguro2.htm

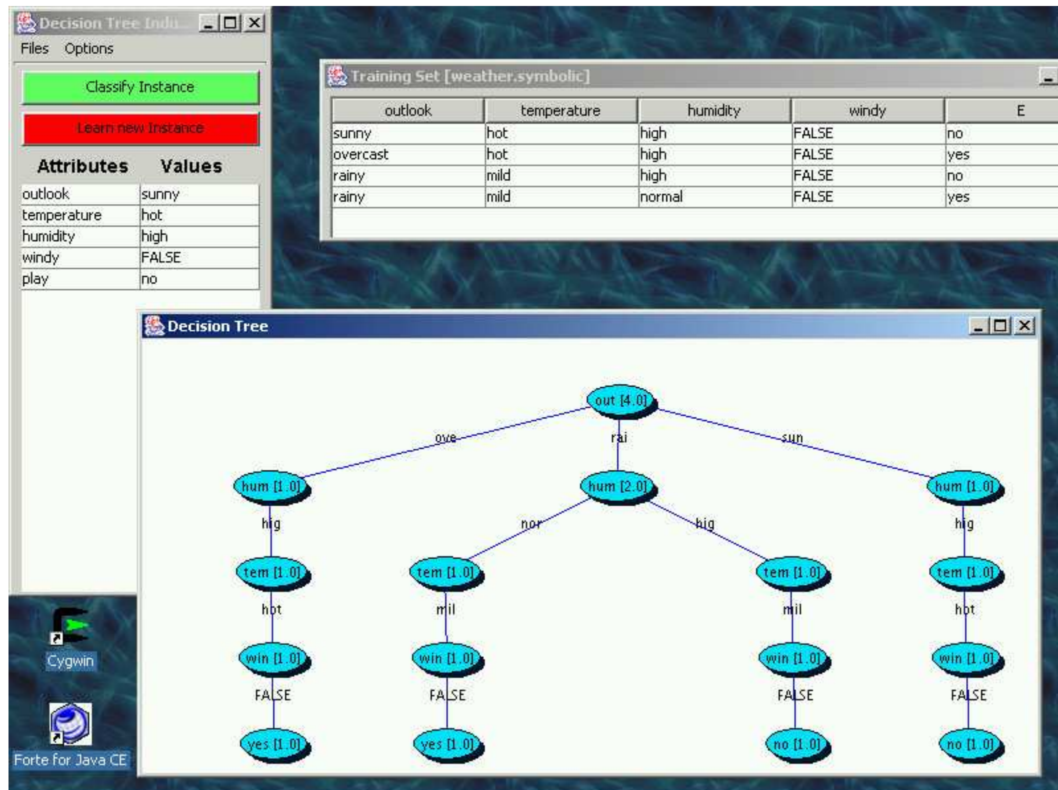


Figura 12.1: Tela principal do xILE

recursos que permitem ao usuário a visualização do caminho na árvore de decisão utilizado na geração de determinada resposta.

13 CONCLUSÕES

13.1 Contribuições

O espectro de contribuições apresentado nesta tese distribui-se equilibradamente entre o domínio da teoria e o da prática, buscando reforçar as inter-relações destes domínios, na proposição de novos formalismos, conceitos, técnicas, ferramentas e aplicações relacionadas com a área dos dispositivos adaptativos. No campo teórico apresentamos dois novos formalismos, os autômatos de estados finitos adaptativos e as árvores de decisão adaptativas. O primeiro destes formalismos foi desenvolvido com finalidades pedagógicas, representando um tipo de dispositivo adaptativo mais simples que os atuais, que pode ser apresentado em uma linguagem "algébrica" mais próxima àquela normalmente encontrada em livros-texto da área da teoria da computação. O segundo formalismo oferece um novo ferramental teórico para especificação de estratégias de aprendizagem de máquina. Junto a este formalismo, apresentamos também um técnica que permite a aplicação da teoria adaptativa em domínios não-discretos e com informações inconsistentes e incompletas, como é o caso de uma grande quantidade de domínios reais.

Ainda no campo teórico, propomos algumas simplificações e uma nova formalização para funções adaptativas, baseada em um subconjunto restrito do cálculo de predicados: a satisfação seqüencial de restrições. Esta nova formalização, por ter sido produzida paralelamente à implementação de uma máquina virtual capaz de executá-la, ajudou a resolver algumas questões de ordem prática que acabaram não sendo suficientemente detalhadas na definição original. Essa máquina virtual foi incorporada a um ambiente gráfico, o AdapTools, de apoio ao desenvolvimento de autômatos adaptativos, que inclui ainda recursos de depuração, simulação, animação e controle de projetos, entre outros.

Uma outra ferramenta apresentada neste trabalho consiste de uma biblioteca de classes Java de apoio ao desenvolvimento de sistemas que integram técnicas de aprendizagem de máquina, como as baseadas em árvores de decisão adaptativas, e processamento digital de sinais, para oferecer uma interface guiada por sinais visuais.

Estas duas ferramentas foram utilizadas no projeto e implementação de uma diversidade de soluções, e protótipos de soluções, para problemas relacionados com construção de compiladores, inteligência artificial, aprendizagem automática, interfaces homem-máquina, visão computacional, diagnóstico médico e tradução texto-voz.

As aplicações incluem um editor de textos que pode ser acionado através de sinais alfabéticos da língua brasileira de sinais; o protótipo de um jogo da velha que pode ser jogado através do olhar; um meta-reconhecedor Wirth-para-AdapTools capaz de produzir analisadores sintáticos com recuperação de erro adaptativa; o protótipo de um tradutor texto-voz bastante simples que tem como núcleo um autômato de estados finitos e um software para a animação de algoritmos de indução de árvores de decisão. Por fim, diversos experimentos controlados foram realizados, apontando que as soluções para aprendizagem de máquina propostas nesta tese possui desempenho comparável, e algumas vezes superior, ao de algoritmos já consolidadas e amplamente utilizados.

13.2 Sugestões para Trabalhos Futuros

Os trabalhos desenvolvidos nesta tese abrem uma série de oportunidades para novas pesquisas na área dos dispositivos adaptativos. Os autômatos de estados finitos adaptativos descrito no capítulo 7, por exemplo, representam um primeiro passo na criação de dispositivos adaptativos com finalidades pedagógicas. Uma questão interessante a ser respondida no futuro é a seguinte: qual o dispositivo adaptativo mais simples que ainda preserva poder de máquina de Turing ? Os próprios autômatos de estados finitos adaptativos precisam ser ainda investigados com maior profundidade, para que possamos obter um prova formal de sua capacidade expressiva e uma formalização mais próxima àquela proposta em (NETO, 2001).

A formalização de funções adaptativas do capítulo 6 também abre um novo espaço para pesquisas, entre as quais, a criação de novos algoritmos, mais eficientes no tempo e no espaço, que o algoritmo 6.1. Estudos mais aprofundados sobre a utilização de recursos mais complexos do cálculo de predicados na especificação de ações elementares

de consulta também deverão ser realizados no futuros. No modelo atual, construções importantes, como a negação de literais e a utilização do operador lógico disjuntivo ("ou"), só podem obtidas indiretamente, através de recursos do mecanismo subjacente. É claro que a adição de recursos mais complexos na camada adaptativa deve ser sempre estudada com cuidado para não torná-la demasiadamente complexa, dificultando assim a sua aceitação.

As árvores de decisão adaptativas estendidas, AdapTree-E, representam um passo importante na ampliação da aplicabilidade dos dispositivos adaptativos para domínios envolvendo valores contínuos, inconsistentes e ausentes. No entanto, a proposta aqui apresentada é apenas uma técnica que permite a união de diferentes estratégias, na construção de sistemas de aprendizagem de máquinas mais poderosos (capazes de lidar tanto com valores discretos quanto contínuos). Seria interessante estudar as possibilidades de criação de modelos e formalismos que permitam uma conceitualização mais homogênea e elegante para dispositivos adaptativos capazes de trabalhar com valores contínuos. Uma linha que nos parece promissora é a da utilização de autômatos híbridos (HENZINGER, 1996; HENZINGER; HO, 1993) como mecanismo subjacente de um novo dispositivo adaptativo. Autômatos híbridos misturam características contínuas e discretas para permitir a modelagem de sistemas constituídos de elementos analógicos e digitais. Acreditamos que também a camada adaptativa tenha que sofrer alterações para facilitar o trabalho com mecanismos subjacentes envolvendo valores contínuos.

Entre os avanços tecnológicos que poderão ser perseguidos no futuro destacamos a otimização da máquina virtual do AdapTools (cap. 8), bem como o projeto e implementação de uma interface ainda mais amigável, com opções que permitam a adequação da ferramenta à diferentes domínios de aplicação. As bibliotecas para o desenvolvimento de sistemas guiados por sinais visuais (cap. 9) poderão vir a ser integradas ao AdapTools, juntamente com outras bibliotecas especializadas para outras áreas. Finalmente, as aplicações descritas nos capítulos 10, 11 e 12 introduzem o uso da tecnologia adaptativa em algumas áreas ainda pouco exploradas, como a de processamento digital de sinais e a da interface homem-máquina. Acreditamos que significativas contribuições científicas e tecnológicas poderão ainda ser obtidas na interação entre estas áreas e a área dos dispositivos adaptativos.

Anexo A - Autômatos de Pilha Estruturados

Um autômato de pilha estruturado, APE, é um sistema $M = (S, Q, \mu, \Sigma, \Gamma, Q_0, F, \delta^i, \delta^e)$ em que:

S um conjunto finito e não vazio de identificadores de sub-máquinas.

Q um conjunto finito e não vazio de estados.

$\mu : Q \rightarrow S$ é uma função que determina, indiretamente, o conjunto de estados de cada uma das sub-máquinas em S. Utilizando o conjunto potência de Q, 2^Q, como contradomínio, obtemos uma função inversa para μ, $\mu' : S \rightarrow 2^Q$, que mapeia cada elemento de S ao seu conjunto de estados. Portanto, dado um $s \in S$, $\mu'(s) \subseteq Q$ é o conjunto de estados de s.

Σ é o alfabeto de entrada do dispositivo.

$Q_0 \subseteq Q$ é um conjunto de estados iniciais em que, $\forall s \in S$, $|\mu'(s) \cap Q_0| = 1$ (Cada sub-máquina possui um, e apenas um, estado inicial)

δ^i é uma relação sobre $\mu'(s) \times \Sigma \cup \{\epsilon\} \times \mu'(s)$, na qual $s \in S$. Cada elemento desta relação é chamado de *transição interna*.

δ^e é uma relação sobre $\mu'(s) \times S \times \mu'(s)$. Cada elemento $(q, m, q') \in \delta^e$ é denominado *transição externa* ou *chamada de sub-máquina*. Os estados q e q' desta transição pertencem a sub-máquina de origem, com q' sendo o "estado de retorno da chamada", enquanto m refere-se à sub-máquina de destino. Durante a operação do autômato o estado de retorno, q', é automaticamente armazenado em uma pilha.

F é o conjunto de estados finais, que correspondem também aos retornos de sub-máquinas. O estado para o qual o controle do autômato deve ser posicionado, após o retorno, é retirado da pilha acima mencionada.

Uma configuração de uma APE é uma tripla (q, x, z), na qual $q \in Q$ é o estado corrente, $x \in \Sigma^*$ é a parte da cadeia de entrada ainda não lida e $z \in Q^*$ é o conteúdo da pilha de estados de retorno de sub-máquina. Quando $\delta^e = \emptyset$ e $|S| = 1$ o APE se especializa em um símples ϵ-AEF, operando como tal. Caso contrário, a relação de passo do autômato, $\vdash$, deve ser extendida para poder representar também os movimentos de chamada e retorno de sub-máquina. Formalmente, dado um $M = (S, Q, \mu, \Sigma, \Gamma, Q_0, F, \delta^i, \delta^e)$, $q, q' \in Q$, $\sigma \in \Sigma \cup \{\epsilon\}$, $x \in \Sigma^*$, $y, y' \in Q^*$ temos que $(q, \sigma x, y) \vdash_M (q', x, y')$ se e somente se uma das três seguintes condições se aplica[1]:

1. $(q, \sigma, q') \in \delta^i$, $y = y'$ and $\mu(q) = \mu(q')$. [Transição Interna]
2. $(q, s, p) \in \delta^e$, $q' \in Q_0 \cap \mu'(s)$, $y' = py$, $\mu(q) = \mu(p)$ and $\sigma = \epsilon$. [Chamada de sub-máquina]
3. $q \in F$, $y = q'y'$ and $\sigma = \epsilon$. [Retorno de sub-máquina]

[1]É importante ressaltar que a formalização aqui proposta omite, para tornar a explicação mais simples, um importate recurso da definição apresentada em (NETO, 1993), que é a possibilidade de retorno de um símbolo para a cadeia de entrada.

Anexo B – Código AdapTools para Aprendizagem por Memorização

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	?Store	?sta	?inp	?x	nop	nop	Store.
2	-Store	?sta	?inp	?x	nop	nop	Store.
3	+Store	?sta	?inp	*y	nop	nop	nop
4	+Store	*y	a	*z	nop	nop	Store.
5	+Store	*y	b	*z	nop	nop	Store.
6	+Store	*y	L	*z	nop	nop	.Learn
7	+Store	*y	;	2	nop	N;	nop
8	?Learn	?x	L	?sta	nop	nop	.Learn
9	-Learn	?x	L	?sta	nop	nop	.Learn
10	-Learn	?x	;	2	nop	N;	nop
11	+Learn	?x	eps	?sta	nop	nop	nop
12	+Learn	?sta	;	2	nop	S;	nop
13	S	0	a	1	nop	nop	Store.
14	S	0	b	1	nop	nop	Store.
15	S	0	L	1	nop	nop	.Learn
16	S	2	eps	0	fin	nop	nop

Entrada: aaab;bab;bbbb;aaabL;bab;aaab;bbbb;babL;bab;bbbb;bbbbL;bbbb;

Saída: N;N;N;S;N;S;N;S;S;N;S;S;

Anexo C – Tratamento de Rotinas Semânticas

O programa abaixo define as rotinas semânticas que são automaticamente carregadas pelo AdapTools. Para utilizar estas rotinas basta inserir o nome das mesmas no campo de símbolo de saída do autômato (coluna *outp*). A adição de novas rotinas semânticas é obtida pela inclusão de novos métodos na classe *adaptools.vm.DefaultSemantics* (e recompilação dos códigos-fonte do AdapTools). Para cada nova rotina, é preciso também adicionar um novo teste ao método *execute*, da classe *adaptools.vm.DefaultSemantics*. Este teste faz a ligação entre o conteúdo da coluna *outp*, passado como parâmetro (*name*) para o método *execute*, e o método que implementa a rotina semântica.

```
package adaptools.vm;

import java.util.*;
import javax.swing.*;

public class DefaultSemantics extends adaptools.vm.Semantics  {

  StringBuffer value;

  public Vmds vmds;

  public DefaultSemantics(Vmds vmds) {
    this.vmds = vmds;
        value = new StringBuffer();
   }

   public Token execute(String name) {
```

```
      if(name.equals("_initBuffer")) return(_initBuffer());
          else if(name.equals("_appendToBuffer")) return(_appendToBuffer());
          else if(name.equals("_metaSymbol")) return(_metaSymbol());
          else if(name.equals("_bufferToken")) return(_bufferToken());
          else if(name.startsWith("_token")) return(_token(name));
          else return(null);
    }

    private Token _initBuffer() {
        value = new StringBuffer();
        return null;
    }

    private Token _appendToBuffer() {
        value.append(vmds.curToken().value);
        return null;
    }

    private Token _metaSymbol() {
        return new Token(vmds.curToken().value,vmds.curToken().value);
    }

    private Token _bufferToken() {
        String temp = value.toString();
        _initBuffer();
        return new Token(temp,temp);
    }

    private Token _token(String temp) {
        temp = temp.substring(temp.indexOf('(')+1,temp.indexOf(')'));
        return new Token(temp,temp);
    }
}
```

A estrutura *Token* que é retornada por todas as rotinas semânticas é definida em *Token.java*, da seguinte forma:

```
package adaptools.vm;

public class Token extends Object  {

    public String label;
    public String value;

    public Token(String label, String value) {
                this.label = label;
                this.value = value;
    }
}
```

Ou seja, ela é apenas um par de *strings*. Os comandos *vmds.curToken().label* e *vmds.curToken().value* podem ser utilizados dentro de qualquer rotina semântica para se obter o nome e o valor do símbolo corrente na cadeia de entrada do autômato. Quando a entrada do autômato é apenas uma cadeia de caracteres, o AdapTools automaticamente transforma cada caracter *c* em um token cujos campos, *label* e *value*, contêm o mesmo valor (a cadeia "c"). Agora, quando a entrada do autômato está conectada à saida de outro autômato, as rotinas semânticas podem ser utilizadas para criar *tokens* mais sofisticados.

Segue abaixo uma pequena descrição das cinco rotinas semânticas implementadas em *adaptools.vm.DefaultSemantics* (o *underscore* que antecede os nomes das rotinas é apenas um convenção para facilitar a identificação de rotinas semânticas no código de um autômato) :

_initBuffer Cria um espaço temporário para armazenamento seqüencial de caracteres.

_appendToBuffer Insere o próximo símbolo da cadeia de entrada no espaço temporário de armazenamento.

_bufferToken Devolve, na forma de um *token*, a cadeia armazenada no espaço temporário. Esta rotina também reinicializa o espaço temporário de armazenamento.

_token(x) Recebe como parâmetro uma cadeia de caracteres x, e devolve um *token* com nome e valor iguais a x.

_metaSymbol Devolve o próximo símbolo da cadeia de entrada como um *token* (campos *label* e *value* são preenchidos com o símbolo da cadeia de entrada).

A classe *adaptools.vm.DefaultSemantics* é apenas uma das possíveis especializações da classe abstrata *adaptools.vm.Semantics*. Outras especializações podem ser criadas caso se deseje separar as rotinas semânticas associadas a diferentes autômatos. Para indicar qual a classe que implementa a semântica de determinado autômato (quando esta classe não é a *adaptools.vm.DefaultSemantics*), é necessário alterar o método *getSemanticObject*, da classe *adaptools.vm.Kernel*, e recompilar o sistema. Por exemplo, para associar as rotinas semânticas implementadas por uma determinada classe *adaptools.contrib.MySemantic*, ao autômato chamado "*MyMachine.spa*", deve-se adicionar ao corpo do método *getSemanticObject* o seguinte teste:

```
  if(  vmds.getObjectFileName().indexOf("MyMachine.spa") != -1 )
       return( new adaptools.contrib.MySemantic(vmds) );
```

O código completo do método *getSemanticObject* na versão atual do AdapTools é mostrado abaixo. É importante notar no código abaixo que, para determinar a semântica que deve ser carregada, vários dos testes procuram apenas por uma subcadeia dentro do nome da autômato. Com isto, qualquer autômato cujo nome contenha a cadeia *lex*, por exemplo, terá sua semântica associada à do analisador léxico do compilador Wirth-AdapTools (implementada pela classe *adaptools.exemples.ccl.Semantics*).

```
public Semantics getSemanticObject() {
  if(  vmds.getObjectFileName().indexOf("sintatico") != -1 ) {
```

```
    System.out.println("CC Semantics Loaded");
    return( new adaptools.examples.cc.Semantics(vmds) );
  }
  else if(vmds.getObjectFileName().indexOf("ccr") != -1 ) {
    System.out.println("CC with Error Recovery Semantics Loaded");
    return( new adaptools.examples.ccr.Semantics(vmds) );
  }
  else if(vmds.getObjectFileName().indexOf("rsw") != -1 ) {
    System.out.println("RSW Semantics Loaded");
    return( new adaptools.examples.rsw.Semantics(vmds) );
  }
  else if(  vmds.getObjectFileName().indexOf("lex") != -1 ) {
    System.out.println("Lexico Semantics Loaded");
    return( new adaptools.examples.ccl.Semantics(vmds) );
  }
  else if(  vmds.getObjectFileName().indexOf("talker") != -1 ) {
    System.out.println("Voice Semantics Loaded");
    return( new adaptools.examples.voice.Semantics(vmds,soundPlayer) );
  }
  else if(  vmds.getObjectFileName().indexOf("extraiEventos") != -1 ) {
    System.out.println("extraiEventos Semantics Loaded");
    return( new adaptools.examples.eventos.Semantics(vmds) );
  }
  else {
    System.out.println("Default Semantics Loaded");
    return( new adaptools.vm.DefaultSemantics(vmds) );
  }
}
```

Anexo D – Clonador

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	S	0	eps	1000	nop	nop	initClone(1).
2	S	1000	eps	1001	nop	nop	.cicle
3	S	1001	eps	1002	nop	nop	.clone
4	S	1002	eps	1003	nop	nop	.break
5	S	1003	eps	1004	nop	nop	.continue
6	S	1004	eps	1	nop	nop	limpa(%).limpa(#)
9	S	1	a	2	fin	nop	nop
10	S	3	b	1	nop	nop	nop
...	...	...	...	...	...	...	...
15	S	5	g	3	nop	nop	nop
16	?initClone	%1	?s	?x	?z1	?z2	?z3
17	+initClone	2000	?s	*n	?z1	?z2	?z3
18	+initClone	*n	#	?x	nop	nop	nop
19	+initClone	%1	%	2000	nop	nop	nop
20	?clone	?x	#	?y	nop	nop	nop
21	?clone	?y	?s	?z	?z1	?z2	?z3
22	-clone	?x	#	?y	nop	nop	nop
23	+clone	?x	?s	*n	?z1	?z2	?z3
24	+clone	*n	#	?z	nop	nop	nop
25	+clone	?y	%	?x	nop	nop	nop
26	?continue	?x	#	?y	nop	nop	nop
27	?continue	?y	?s	?z	?z1	?z2	?z3
28	-continue	1004	eps	1	nop	nop	limpa(%).limpa(#)
29	+continue	1004	eps	1000	nop	nop	nop
30	?limpa	?x	%1	?y	nop	nop	nop
31	-limpa	?x	%1	?y	nop	nop	nop
32	?cicle	?x	?s	?y	?z1	?z2	?z3
33	?cicle	?y	#	?z	nop	nop	nop
34	?cicle	?z	%	?w	nop	nop	nop
35	-cicle	?x	?s	?y	?z1	?z2	?z3
36	-cicle	?y	#	?z	nop	nop	nop
37	+cicle	?x	?s	?w	?z1	?z2	?z3
38	?break	1004	eps	1000	nop	nop	nop
39	-break	1004	eps	1000	nop	nop	nop
40	+break	1004	eps	1	nop	nop	limpa(%).limpa(#)

Anexo E – Meta-Compilador Wirth para AdapTools

Código do Analisador Sintático:

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	Gram	0	n	1	nop	rs0	nop
2	Gram	1	=	2	nop	rs2	nop
3	Gram	2	eps	5	3	nop	nop
4	Gram	3	.	4	nop	rs8	nop
5	Gram	4	n	1	nop	rs7	nop
6	Gram	4	eps	pop	nop	fin	nop
7	Expr	5	&	6	nop	rs1	nop
8	Expr	5	t	6	nop	rs1	nop
9	Expr	5	n	6	nop	rs1	nop
10	Expr	5	(	7	nop	rs2	nop
11	Expr	5	[	9	nop	rs3	nop
12	Expr	5	{	11	nop	rs4	nop
13	Expr	6	&	6	nop	rs1	nop
14	Expr	6	t	6	nop	rs1	nop
15	Expr	6	n	6	nop	rs1	nop
16	Expr	6	—	5	nop	rs6	nop
17	Expr	6	(	7	nop	rs2	nop
18	Expr	6	[	9	nop	rs3	nop
19	Expr	6	{	11	nop	rs4	nop
20	Expr	6	eps	pop	nop	nop	nop
21	Expr	7	eps	5	8	nop	nop
22	Expr	8	)	6	nop	rs5	nop
23	Expr	9	eps	5	10	nop	nop
24	Expr	10	]	6	nop	rs5	nop
25	Expr	11	eps	5	12	nop	nop
26	Expr	12	}	6	nop	rs5	nop

Código do Analisador Léxico:

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	Main	0	spc	0	fin	nop	nop
2	Main	0	”	1	nop	_initBuffer	nop
3	Main	0	=	2	nop	_metaSymbol	nop
4	Main	0	—	2	nop	_metaSymbol	nop
5	Main	0	&	2	nop	_metaSymbol	nop
6	Main	0	{	2	nop	_metaSymbol	nop
7	Main	0	}	2	nop	_metaSymbol	nop
8	Main	0	[	2	nop	_metaSymbol	nop
9	Main	0	]	2	nop	_metaSymbol	nop
10	Main	0	(	2	nop	_metaSymbol	nop
11	Main	0	)	2	nop	_metaSymbol	nop
12	Main	0	.	2	nop	_metaSymbol	nop
13	Main	0	eps	100	3	_initBuffer	nop
14	Main	1	”	2	nop	_terminal	nop
15	Main	1	eps	100	1	nop	nop
16	Main	1	eps	200	1	nop	nop
17	Main	3	eps	100	3	nop	nop
18	Main	3	eps	200	3	nop	nop
19	Main	3	eps	2	nop	_nonTerminal	nop
20	Main	2	eps	0	fin	nop	nop
21	Special	1	spc	1	nop	_appendToBuffer	nop
22	Special	1	(	1	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
52	Special	1	.	1	nop	_appendToBuffer	nop
53	Escape	1		10	nop	nop	nop
54	Escape	10		1	nop	_appendToBuffer	nop
55	Escape	10	”	1	nop	_appendToBuffer	nop
56	Digi	200	0	201	nop	_appendToBuffer	nop
57	Digi	200	1	201	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
65	Digi	200	9	201	nop	_appendToBuffer	nop
66	Digi	201	eps	pop	fin	nop	nop
67	Lett	100	a	101	nop	_appendToBuffer	nop
68	Lett	100	b	101	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
118	Lett	100	Z	101	nop	_appendToBuffer	nop
119	Lett	101	eps	pop	fin	nop	nop

Exemplo de Entrada:

```
Lista = "(" Numero { "," Numero } ")"  .

Numero = D { D }.

D = "0"  | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |  "9" .
```

Exemplo de Saída:

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	Lista	0	(	2	nop	nop	nop
2	Lista	2	eps	1000	3	nop	nop
3	Lista	3	eps	4	nop	nop	nop
4	Lista	4	,	5	nop	nop	nop
5	Lista	5	eps	1000	6	nop	nop
6	Lista	6	eps	4	nop	nop	nop
7	Lista	4	)	7	nop	nop	nop
8	Lista	7	eps	1	nop	nop	nop
9	Lista	1	eps	pop	fin	nop	nop
10	Numero	1000	eps	2000	1002	nop	nop
11	Numero	1002	eps	1003	nop	nop	nop
12	Numero	1003	eps	2000	1004	nop	nop
13	Numero	1004	eps	1003	nop	nop	nop
14	Numero	1003	eps	1001	nop	nop	nop
15	Numero	1001	eps	pop	fin	nop	nop
16	D	2000	0	2002	nop	nop	nop
17	D	2002	eps	2001	nop	nop	nop
18	D	2000	1	2003	nop	nop	nop
19	D	2003	eps	2001	nop	nop	nop
20	D	2000	2	2004	nop	nop	nop
21	D	2004	eps	2001	nop	nop	nop
...	...	...	...	...	...	...	...
33	D	2010	eps	2001	nop	nop	nop
34	D	2000	9	2011	nop	nop	nop
35	D	2011	eps	2001	nop	nop	nop
36	D	2001	eps	pop	fin	nop	nop

Anexo F – Um Exemplo de Recuperação de Erros usando Autômatos Adaptativos

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	S	0	a	1	nop	nop	.E1(1,b,2)
2	S	1	b	2	nop	nop	E2.
3	S	2	c	0	fin	nop	nop
4	+E1	$\%_1$	a	9990	nop	[ERRO-Ins	nop
5	+E1	9990	$\%_2$	$\%_3$	nop	]	.E2
6	+E1	$\%_1$	eps	9990	nop	[Erro	nop
7	+E1	9990	eps	$\%_3$	nop	-Rem]	.E2
8	?E2	?x1	?x2	9990	?x3	?x4	?x5
9	?E2	9990	?y1	?y2	?y3	?y4	?y5
10	-E2	?x1	?x2	9990	?x3	?x4	?x5
11	-E2	9990	?y1	?y2	?y3	?y4	?y5

Entrada: abcacaabcaac

Saída: abca[Erro-Rem]ca[ERRO-Ins]ca[ERRO-Ins-Rem]c

Anexo G – Implementação da Semântica do Tradutor Texto-Voz

```
/** Semantics.java

 Provides the semantics for a very simple texto-to speech translator.

 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

 @author Hemerson Pistori (pistori@ec.ucdb.br)

 This work is based on Dr. Joao Jose Neto theories and techniques on
 Automata and Formal Languages (http://www.pcs.usp.br/~lta).


*/
```

```
package adaptools.examples.voice;

import adaptools.vm.*;
import java.util.*;
import javax.swing.*;


public class Semantics extends adaptools.vm.Semantics  {

    public String soundExt = ".wav";
    public String baseDir = "examples//voice//phonemes//";
        public Vmds vmds;
        public SoundPlayer player;

    public Semantics(Vmds vmds, SoundPlayer player) {
                this.vmds = vmds;
                this.player = player;
    }

    public Token execute(String name) {
                return(say(name));
    }

    private Token say(String name) {
                player.playAndWait(baseDir+name+soundExt);
                return(null);
    }

}
```

Referências Bibliográficas

AHA, D. W.; KIBLER, D.; ALBERT, M. Instance-based learning algorithms. *Machine Learning*, v. 6, p. 37–66, 1991.

AHO, A. V.; PETERSON, T. G. A minimum distance error-correcting parser for context-free languages. *SIAM Jornal on Computing*, v. 1, n. 4, p. 305–312, 1972.

AL-JARRAH, O.; HALAWANI, A. Recognition of gestures in arabic sign language using neuro-fuzzy systems. *Artificial Intelligence*, Elsevier Science, v. 133, p. 117–138, December 2001.

ANDERSSON, A.; DAVIDSSON, P.; LIND'EN, J. *Measuring generalization quality*. 1998.

BALUJA, S.; POMERLEAU, D. Non-intrusive gaze tracking using artificial neural networks. In: COWAN, J. D.; TESAURO, G.; ALSPECTOR, J. (Ed.). *Advances in Neural Information Processing Systems*. Morgan Kaufmann Publishers Inc., 1994. v. 6, p. 753–760. Disponível em: <citeseer.nj.nec.com/baluja94nonintrusive.html>.

BASSETO, B. A.; NETO, J. J. A stochastic musical composer based on adaptative algorithms. In: *Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação. SBC-99*. PUC-RIO, Rio de Janeiro, Brazil: [s.n.], 1999. v. 3, p. 105–130.

BLAKE, C.; MERZ, C. UCI - repository of machine learning databases. *University of California, Irvine, Dept. of Information and Computer Sciences*, 1998. Disponível em: <http://www.ics.uci.edu/~mlearn/MLRepository.html>.

BOULLIER, P. *Dynamic Grammars and Semantic Analysis*. [S.l.], august 1994.

BRAFFORT, A. Research on computer science and sign language: Ethical aspects. In: *Gesture Workshop*. [s.n.], 2001. p. 1–8. Disponível em: <citeseer.nj.nec.com/braffort01research.html>.

BUCHANAN, B.; SHORTLIFFE, E. H. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.

BURSHTEYN, B. Generation and recognition of formal languages by modifiable grammars. *ACM SIGPLAN Notices*, v. 25, n. 12, p. 45–53, 1990.

BURSHTEYN, B. On the modification of the formal grammar at parse time. *ACM SIGPLAN Notices*, v. 25, n. 5, p. 117–23, 1990.

CABASINO, S.; PAOLUCCI, P. S.; TODESCO, G. M. Dynamic parsers and evolving grammars. *ACM SIGPLAN Notices*, v. 27, n. 11, p. 39–48, 1992.

CAMOLESI, A. R.; NETO, J. J. Modelagem AMBER-Adp de um ambiente para gerenciamento de ensino a distância. In: *Anais do XIII Simpósio Brasileiro de Informática na Educação. SBIE 2002*. São Leopoldo, RS: [s.n.], 2002. p. 401–409.

CAPOVILLA, F. C.; RAPHAEL, W. D. *Dicionário Enciclopédico Ilustrado Trilíngue da Língua de Sinais Brasileira*. São Paulo, Brasil: Editora da Universidade de São Paulo, 2001.

CHRISTIANSEN, H. Recognition of generative languages. *Lecture Notes in Computer Science*, New York: Springer-Verlag, n. 217, p. 63–81, 1986.

CHRISTIANSEN, H. Why should grammars not adapt themselves to context and discourse? *Abstract collection, 4th International Pragmatics Conference*, International Pragmatics Association, Kobe, Japan, july 1993.

CHRISTIANSEN, H.; SHAW, C. J. A survey of adaptable grammars. *ACM SIGPLAN Notices*, v. 25, n. 11, p. 35–44, 1990.

CLARK, P.; NIBLETT, T. The CN2 induction algorithm. *Machine Learning*, v. 3, p. 261–283, 1989.

CODD, E. F. A relational model of data for large shared data bases. *Communications of ACM*, v. 13, n. 6, p. 377–387, 1970.

COLOMBO, C.; BIMBO, A. D. Interacting through eyes. *Robotics and Autonomous Systems*, v. 19, p. 359–368, 1997.

COSTA, E. R.; HIRAKAWA, A. R.; NETO, J. J. An adaptive alternative for syntactic pattern recognition. In: *Proceeding of 3rd International Symposium on Robotics and Automation, ISRA*. Toluca, Mexico: [s.n.], 2002. p. 409–413.

CUNNINGHAM, S. J.; HOLMES, G. Developing innovative applications of machine learning. *Proc. of Southeast Asia Regional Computer Confederation Conference*, Singapore, 1999.

DAVIS, J.; SHAH, M. Recognizing hand gestures. In: *Proceedgins of the Third European Conference on Computer Vision (ECCV)*. Stockholm, Sweden: [s.n.], 1994. p. 331–340.

DOMINGOS, P.; PAZZANI, M. J. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In: *International Conference on Machine Learning*. [S.l.: s.n.], 1996. p. 105–112.

DOUGHERTY, J.; KOHAVI, R.; SAHAMI, M. Supervised and unsupervised discretization of continuous features. In: PRIEDITIS, A.; RUSSELL, S. (Ed.). *Machine Learning: Proceedings of the Twelfth Conference*. [S.l.]: Morgan Kaufmann Publishers, 1995. p. 194–202.

ERDEM, U. M.; SCLAROFF, S. Automatic detection of relevant head gestures in american sign language communication. In: *Proceedings of the International Conference on Pattern Recognition - ICPR 2002*. Québec, Canada: [s.n.], 2002. Disponível em: <citeseer.nj.nec.com/508432.html>.

FANG, G. et al. Signer-independent continuous sign language recognition based on SRN/HMM. In: *Gesture Workshop*. [s.n.], 2001. p. 76–85. Disponível em: <citeseer.nj.nec.com/502642.html>.

FAYYAD, U. M.; IRANI, K. B. Multi-interval discretization of continuous-valued attributes for classification learning. In: *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann, 1993. p. 1022–1027.

FORINO, A. C. d. Some remarks on the syntax of symbolic programming languages. *Communications of the ACM*, v. 6, p. 456–460, 1963.

FORSTER, M. R. The new science of simplicity. In: ____. *Simplicity, Inference and Modelling*. [S.l.]: Cambridge University Press, 2001. cap. 6, p. 83–119.

FRANK, E. et al. Using model trees for classification. *Machine Learning*, v. 32, n. 1, p. 63–76, 1998.

FREEMAN, W. T. et al. Computer vision for interactive computer graphics. *IEEE Computer Graphics and Applications*, v. 18, n. 3, p. 42–53, /1998. Disponível em: <citeseer.nj.nec.com/freeman98computer.html>.

FREEMAN, W. T. et al. Computer vision for computer games. In: *2nd International Conference on Automatic Face and Gesture Recognition*. Killington, VT, USA: [s.n.], 1996. p. 100–105.

FREITAS, A. V.; NETO, J. J. Aspectos do projeto e implementação de ambientes multiparadigmas de programação. In: *Proceedings of ICIE Y2K - International Congress on Informatics Engineering*. Buenos Aires, Argentina: [s.n.], 2000.

FREITAS, A. V.; NETO, J. J. Uma ferramenta para construção de aplicações multilinguagens de programação. In: *CACIC'2001 - Congreso Argentino de Ciencias de la Computación*. El Calafate, Argentina: [s.n.], 2001.

GÖDEL, K. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. [S.l.]: Dover Publications, 1992.

GEE, A. H.; CIPOLLA, R. *Determining the Gaze of Face in Images*. Trumpington Street, Cambridge CB2 1PZ, England, 1994. Disponível em: <citeseer.nj.nec.com/gee94determining.html>.

GENESERETH, M. R.; NILSSON, N. J. *Logical Foundations of Artificial Intelligence*. [S.l.]: Morgan Kaufmann, 1988.

GILES, C. L. et al. Constructive learning of recurrent neural networks: Limitations of recurrent cascade correlation and a simple solution. *IEEE Transactions on Neural Networks*, v. 6, n. 4, p. 829–836, July 1995.

GOMEZ M. SANCHEZ, L. E. S. G. On selecting an appropriate colour space for skin detection. *MICAI 2002: Advances in Artificial Intelligence, Second Mexican International Conference on Artificial Intelligence. Lecture Notes in Artificial Intelligence*, Elsevier-Science, p. 69–78, 2002.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. [S.l.]: Addison-Wesley Pub. Co., 2002.

GRUNE, D.; JACOBS, C. J. H. *Parsing techniques a practical guide*. Chichester, England: Ellis Horwood Limited, 1990. Disponível em: <citeseer.nj.nec.com/grune90parsing.html>.

HANFORD, K.; JONES, C. Dynamic syntax: A concept for the definition of the syntax of programming languages. *Annual Review in Automatic Programming*, Pergamon Press, Oxford, v. 7, p. 115–142, 1973.

HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, v. 8, n. 3, p. 231–274, June 1987. Disponível em: <citeseer.nj.nec.com/article/harel87statecharts.html>.

HAYKIN, S. *Neural Networks. A Comprehensive Foundation*. New Jersey, USA: Prentice Hall, 1999.

HECKERMAN, D.; GEIGER, D.; CHICKERING, D. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, v. 20, p. 197–243, 1995.

HEDBERG, S. R. Stanford university's ai in medicine: Still cutting the edge. *IEEE Intelligente Systems*, v. 13, n. 1, p. 74–76, 1998.

HENZINGER, R. A. C. C. T. A.; HO, P.-H. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science*, Springer-Verlag, v. 736, p. 209–229, 1993.

HENZINGER, T. The theory of hybrid automata. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. New Brunswick, New Jersey: [s.n.], 1996. p. 278–292. Disponível em: <citeseer.nj.nec.com/henzinger96theory.html>.

HIRSCHBERG, D. S.; LARMORE, L. L. New applications of failure functions. *Journal of the ACM*, v. 34, n. 3, p. 616–625, 1987. Disponível em: <citeseer.nj.nec.com/hirschberg87new.html>.

IWAI, M. K. *Um Formalismo Gramatical Adaptativo para Linguagens Dependentes de Contexto*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 2000.

IWAI, M. K.; NETO, J. J. *Introdução às Gramáticas Adaptativas*. São Paulo, Brasil, 2000.

JACKSON, Q. T. Adaptive predicates in natural language parsing. *Perfection*, n. 4, 2000.

JACKSON, Q. T. Efficient formalism-only parsing of xml/html using the §-calculus. *Noesis-E*, v. 2, n. 2, june 2002.

JACKSON, Q. T.; LANGAN, C. M. Adaptive predicates in empty-start natural language parsing. *Noesis-E*, v. 1, n. 6, November 2001.

JACOB, R. Eye tracking in advanced interface design. *In Barøeld,W. and Furness, T. (Eds.), Advanced Interface Design and Virtual Environments*, Oxford University Press, p. 258–288, 1995. Disponível em: <citeseer.nj.nec.com/jacob95eye.html>.

JUNIOR, J. R. A. *Stad :uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos.* Tese (Doutorado) — Escola Politécnica, Universidade de Sao Paulo, São Paulo, Brasil, 1995.

JUNIOR, J. R. A.; NETO, J. J.; HIRAKAWA, A. R. Adaptive automata for independent autonomous nagivation in unknown enviroment. In: *Proceedings of IASTED International Conference on Applied Simulation and Modelling - ASM 2000*. Banff, Alberta: [s.n.], 2000.

KANDOLA J.S. E GUNN, S. Assessing the stability of advanced transparent modelling techniques. In: *Proceedings CRM Workshop on Combining and Selecting Models using Machine Learning Algorithms*. Montreal, Canada: [s.n.], 2000.

KAPUSCINSKI, T.; WYSOCKI. Hand gesture recognition for man-machine interaction. In: *Proceedings of the Second International Workshop on Robot Motion and Control*. Bukowy Dworek, Poland: [s.n.], 2001. p. 91–96.

KIDA, T. et al. Multiple pattern matching in LZW compressed text. In: *Proc. of Data Compression Conference*. IEEE Computer Society, 1998. p. 103–112. Disponível em: <citeseer.nj.nec.com/article/kida98multiple.html>.

KOHAVI, R.; SAHAMI, M. Error-based and entropy-based discretization of continuous features. In: *International Conference on Knowledge and Data Mining*. [S.l.: s.n.], 1996.

KOTHARI, R.; DONG, M. Pattern recognition: From classical to modern approaches. In: ____. [S.l.]: World Scientific, 2001. cap. 6, Decision Trees For Classification: A Review and Some New Results, p. 169–184.

L.BRETZNER; I.LAPTEV; T.LINDEBERG. Hand gesture recognition using multi-scale colour features, hierarchical models and particle filtering. In: *Proceedings of the Fifth IEEE International Conference on Automatic Face and Gesture Recognition (FGR'02)*. Washington, USA: [s.n.], 2002. p. 423–428.

LEDENIOV, O.; MARKOVITCH, S. The divide-and-conquer subgoal-ordering algorithm for speeding up logic inference. *Journal of Artificial Intelligence Research*, v. 9, p. 37–97, 1998.

LEVINE, J. R.; MASON, T.; BROWN, D. *Lex and Yacc*. [S.l.]: O'Reilly, 1992.

MARCATO, S. A.; ROCHA, H. V. da; LIMA, M. C. M. P. O uso da internet no aprendizado da língua de sinais. In: *Anais do IBERDISCAP*. Madrid, Espanha: [s.n.], 2000.

MARTIN, J.; CROWLEY, J. L. An appearance-based approach to gesture recognition. In: *Proc. of the 9th Int. Conf. on Image Analysis and Processing*. Florence, Italy: [s.n.], 1997.

MARTIN, J.; DEVIN, V.; CROWLEY, J. Active hand tracking. In: *In IEEE Third International Conference on Automatic Face and Gesture Recognition, FG '98*. Nara, Japan: [s.n.], 1998. Disponível em: <citeseer.nj.nec.com/martin98active.html>.

MARTíNEZ, A. et al. Purdue RVL-SLLL ASL database for automatic recognition of american sign language. In: *Proc. IEEE International Conference on Multimodal Interfaces*. [S.l.: s.n.], 2002.

MASON, K. P. *Dynamic Template Translators: A useful model for the definition of programming languages*. Tese (Doutorado) — University of Adelaide, Adelaide, Australia, 1984.

MASON, K. P. Dynamic template translators - a new device for specifying programming languages. *International Journal of Computer Math*, v. 22, p. 199–212, 1987.

MAUNEY, J.; FISHER, C. N. A forward move algorithm for ll and lr parsers. *SIGPLAN Notices*, v. 17, n. 6, p. 79–87, 1982.

MCLAUGHLIN, R. A. Randomized hough transform: Improved ellipse detection with comparison. *Pattern Recognition Letters*, v. 19, n. 3, p. 299–305, 1998. Disponível em: <citeseer.nj.nec.com/mclaughlin98randomized.html>.

MENEZES, C.; NETO, J. J. Um método para a construção de analisadores morfológicos, aplicado à lıingua portuguesa, baseado em autômatos adaptativos. In: *V PROPOR, Encontro para o Processamento Computacional de Português Escrito e Falado*. Atibaia, Brasil: [s.n.], 2000.

MENEZES, C. E. D. de. *Um Método para a Construção de Analisadores Morfológicos, Aplicado à Lingua Portuguesa, Baseado em Autômatos Adaptativos*. Dissertação (Mestrado) — Escola Politécnica, Universidade de Sao Paulo, São Paulo, Brasil, Julho 2000.

MENEZES, C. E. D. de; NETO, J. J. Um método hıibrido para a construção de etiquetadores morfológicos, aplicado à lıingua portuguesa, baseado em autômatos adaptativos. In: *Anais da Conferencia Iberoamericana en Sistemas, Cibernética e Informática*. Orlando, Florida: [s.n.], 2002.

MICHALSKI, R.; BRATKO, I.; KUBAT, M. *Machine Learning and Data Mining - Methods and Applications*. [S.l.]: John Wiley and Sons, 1998.

MINGERS, J. Expert systems - rule induction with statistical data. *Journal of the Operational Research Society*, v. 38, p. 39–47, 1987.

MINKER, J. Logic and databases: A 20 year retrospective. In: *Logic in Databases*. [s.n.], 1996. p. 3–57. Disponível em: <citeseer.nj.nec.com/minker96logic.html>.

MITCHELL, T. M. *Machine Learning*. [S.l.]: McGraw-Hill, 1997.

MOHRI, M. String-matching with automata. *Nordic Journal of Computing*, v. 4, n. 2, p. 217–231, Summer 1997. Disponível em: <citeseer.nj.nec.com/mohri97stringmatching.html>.

MOHRI, M. Edit-distance of weighted automata. *Conference on Implementation and Application of Automata - CIAA 2002*, Tours, France, p. 7–29, July 2002.

MUGGLETON, S.; RAEDT, L. D. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, v. 19/20, p. 629–679, 1994.

NETO, J. J. A automação do projeto e desenvolvimento de sistemas digitais: O SPD como ferramenta de obtenção automática de *Cross-Software* para sistemas de computação. In: *Anais da IV Congresso Brasileiro de Automática*. Campinas, SP: [s.n.], 1982. p. 5252–530.

NETO, J. J. *Cross-Assemblers* para microprocessadores - geração automática através do spd. In: *Anais do I CONAI - Congresso Nacional de Automação Industrial*. São Paulo, SP: [s.n.], 1983. p. 501–509.

NETO, J. J. Geração automática de analisadores sintáticos para o spd - evolução e estado da arte. In: *Anais do VI Congresso Nacional de Matemática Aplicada e Computação*. São José dos Campos, SP: [s.n.], 1983.

NETO, J. J. *Introdução à Compilação*. Rio de Janeiro: LTC, 1987. 222 p.

NETO, J. J. *Compilador de Gramáticas Descritas em Notação de Wirth Modificada*. São Paulo, Brasil, 1988.

NETO, J. J. *Uma Solução Adaptativa para Reconhecedores Sintáticos*. São Paulo, Brasil, 1988.

NETO, J. J. Contribuição à metodologia de construção de compiladores. *Post-Doctoral Tese de Livre Docência, Escola Politécnica, Universidade de São Paulo*, São Paulo, Brasil, 1993.

NETO, J. J. Adaptive automata for context-sensitive languages. *SIGPLAN NOTICES*, v. 29, n. 9, p. 115–124, September 1994.

NETO, J. J. Solving complex problems efficiently with adaptative automata. In: *Conference on the Implementation and Application of Automata - CIAA 2000*. Ontario, Canada: [s.n.], 2000.

NETO, J. J. Adaptative rule-driven devices - general formulation anda case study. In: *CIAA'2001 Sixth International Conference on Implementation and Application of Automata*. Pretoria, South Africa: [s.n.], 2001. p. 234–250.

NETO, J. J.; FREITAS, A. V. Using adaptive automata in a multi-paradigm programming environment. In: *ASM2001 - IASTED International Conference on Applied Simulation and Modelling*. Marbella, Espanha: [s.n.], 2001.

NETO, J. J.; IWAI, M. K. Adaptive automata for syntax learning. In: *Anais da XXIV Conferencia Latinoamericana de Informática - CLEI 98*. Quito, Equador: [s.n.], 1998. p. 135–149.

NETO, J. J.; JUNIOR, J. R. A. Modeling adaptive reactive systems. In: *International Conference on Applied Modelling and Simulation*. Cairns, Australia: [s.n.], 1999.

NETO, J. J.; JUNIOR, J. R. A. Using adaptive models for systems descripton. In: *International Conference on Applied Modelling and Simulation*. Cairns, Australia: [s.n.], 1999.

NETO, J. J.; JUNIOR, J. R. A.; SANTOS, J. M. N. dos. Synchronized statecharts for reactive systems. In: *Proceedings of the IASTED International Conference on Applied Modelling and Simulation*. Honolulu, Hawaii: [s.n.], 1998. p. 246–251.

NETO, J. J.; MAGALHÃES, M. E. Reconhecedores sintáticos uma alternativa didática para uso em cursos de engenharia. In: *Anais da XIV Congresso Nacional de Informática - SUCESU*. São Paulo, SP: [s.n.], 1981. p. 171–181.

NETO, J. J.; MAGALHÃES, M. E. Um gerador automático de reconhecedores sintáticos para o spd. In: *Anais da VIII SEMISH - Seminário de Software e Hardware*. [S.l.: s.n.], 1981. p. 213–229.

NETO, J. J.; MAGALHÃES, M. E. Um suporte para a geração de compiladores. In: *Anais da XVI Congresso Nacional de Informática - SUCESU*. São Paulo, SP: [s.n.], 1983. p. 140–143.

NETO, J. J.; MORAES, M. de. Formalismo adaptativo aplicado ao reconhecimento de linguagem natural. *Anais da Conferencia Iberoamericana en Sistemas, Cibernética e Informática*, Orlando, Florida, July 2002.

NETO, J. J.; PARIENTE, C. A. B. Adaptative automata - a reduced complexity proposal. In: *Proceedings of the Conference on the Implementation and Application of Automata - CIAA 2002*. Tours, France: [s.n.], 2002. p. 161–170.

NETO, J. J.; PARIENTE, C. B.; LEONARDI, F. Compiler construction - a pedagogical approach. In: *Proceedings of the V International Congress on Informatic Engineering - ICIE 99*. Buenos Aires, Argentina: [s.n.], 1999.

OMLIN, C.; GILES, C. L. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, v. 9, n. 1, p. 41, 1996.

PAREKH, R.; HONAVAR, V. Dfa learning from simple examples. *Machine Learning*, v. 44, p. 9–35, 2001.

PATERSON, M.; WEGMAN, M. Linear unification. *Journal of Computer and System Sciences*, v. 16, p. 158–167, 1978.

PEREIRA, J. C. D.; NETO, J. J. Um ambiente de desenvolvimento de reconhecedores sintáticos baseado em autômatos adaptativos. In: *Anais do II Simpósio Brasileiro de Linguagens de Programação - SBLP97*. Campinas, Brasil: [s.n.], 1997. p. 139–150.

PICCHIA, W. D. *Métodos numéricos para a resolução de problemas lógicos*. [S.l.]: Edgar Blucher, 1993.

PILU, A. F. M.; FISHER, R. Ellipse-specific direct least-square fitting. *IEEE International Conference on Image Processing*, September 1996.

PISTORI, H. Portabilidade de aplicativos com interface gráfica. *Anais da Mostra Científica do XXXIII Congresso Internacional de Informática e Telecomunicações - SUCESU 2000*, Campo Grande, MS, 2000.

PISTORI, H.; NETO, J. J. Adaptree - proposta de um algoritmo para indução de Árvores de decisão baseado em técnicas adaptativas. In: *Anais da Conferência Latino Americana de Informática - CLEI 2002*. Montevideo, Uruguai: [s.n.], 2002.

PISTORI, H.; NETO, J. J. A free software for the development of adaptive automata. In: *Proceedings of the IV Workshop on Free Software - WSL (IV International Forum on Free Software)*. Porto Alegre, Brasil: [s.n.], 2003.

PISTORI, H.; NETO, J. J.; COSTA, E. R. Utilização de tecnologia adaptativa na detecção da direção do olhar. In: *Anais da Conferencia Internacional de la Sociedad Peruana de la Computación SPC'2003*. Lima, Peru: [s.n.], 2003.

PISTORI, H.; NETO, J. J.; COSTA, E. R. Utilização de tecnologia adaptativa na detecção da direção do olhar (completo). *SPC Magazine*, Lima, Peru, v. 2, n. 2, Maio 2003.

PORILL, J. Fitting ellipses and predicting confidence envelopes using a bias corrected Kalman filter. *Image and Vision Computing*, v. 8, n. 1, p. 37–41, 1990.

QUINLAN, J. R. Unknown attribute values in induction. In: SEGRE, A. (Ed.). *Proceedings of the Sixth International Machine Learning Workshop*. New York, USA: Morgan Kaufmann Publishers, 1989.

QUINLAN, J. R. Learning with Continuous Classes. In: *5th Australian Joint Conference on Artificial Intelligence*. [S.l.: s.n.], 1992. p. 343–348.

QUINLAN, J. R. *C4.5 Programs for Machine Learning*. [S.l.]: Morgan Kaufmann, 1993.

QUINLAN, J. R. Learning decision tree classifiers. *ACM Computing Surveys*, San Mateo, CA, v. 28, n. 1, 1996.

RIDLER, T. W.; CALVARD, S. Picture thresholding using an iterative selection method. *IEEE transactions on Systems, Man and Cybernetics*, August 1978.

RISSANEN, J. Modeling by shortest data description. *Automatica*, v. 14, p. 465–471, 1978.

ROBINSON, J. A machine-oriented logic based on the resolution principle. *Journal of ACM*, v. 12, p. 23–41, 1965.

ROCHA, R. L. A. Uma proposta de uso de tecnologia adaptativa para simulação de redes neurais em um dispositivo computacional. In: *Proceedings of the IX Encuentro Chileno de Computación*. Punta Arenas, Chile: [s.n.], 2001. p. 1–9.

ROCHA, R. L. A.; NETO, J. J. Autômato adaptativo, limites e complexidade em comparação com máquina de turing. In: *Proceedings of the second Congress of Logic Applied to Technology - LAPTEC'2000*. São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia: [s.n.], 2000. p. 33–48.

ROCHA, R. L. A.; NETO, J. J. Construction of models based on adaptive automata through grammar descriptions. In: *ASM'2000 - International Conference on Applied Simulation and Modelling*. Banff, Canada: [s.n.], 2000. p. 228–234.

ROCHA, R. L. A.; NETO, J. J. Uma proposta de método adaptativo para a seleção automática de soluções. In: *Proceedings of ICIE Y2K - International Congress on Informatics Engineering*. Buenos Aires, Argentina: [s.n.], 2000.

ROCHA, R. L. A.; NETO, J. J. Construção e simulação de modelos baseados em autômatos adaptativos em linguagem funcional. In: *Proceedings of ICIE 2001 - International Congress on Informatics Engineering*. Buenos Aires, Argentina: [s.n.], 2001. p. 509–521.

ROCHA, R. L. A.; NETO, J. J. A proposal of a computational device based on adaptive automata. In: *Proceedings of the second Congress of Logic Applied to Technology - LAPTEC'2001*. São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia, Brasil: [s.n.], 2001. II, p. 145–152.

RUBINSTEIN, R.; SHUTT, J. N. *Self-Modifying Finite Automata*. Worcester, Massachusetts, December 1993.

RUBINSTEIN, R.; SHUTT, J. N. Self-modifying finite automata. In: *Proceeding of the 13th IFIP World Computer Congress*. Amsterdam: North-Holland: [s.n.], 1994. p. 493–498.

RUBINSTEIN, R. S.; SHUTT, J. N. Self-modifying finite automata: An introduction. *Information Processing Letters*, v. 56, n. 4, p. 185–190, 1995.

SANTOS, J. M. N. d. *Um formalismo adaptativo com mecanismo de sincronização para aplicações concorrentes.* Dissertação (Mestrado) — Escola Politécnica, Universidade de Sao Paulo, São Paulo, Brasil, 1997.

SATO, Y.; KOBAYASHI, Y.; KOIKE, H. Fast tracking of hands and fingertips in infrared images for augmented desk interface. In: *Fourth International Conference on Automatic Face- and Gesture-Recognition*. Grenoble, France: [s.n.], 2000.

SCHAFFER, C. Overfitting avoidance as bias. In: *IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*. Sydney, Australia: [s.n.], 1991.

SCHAFFER, C. A conservation law for generalization performance. In: COHEN, W.; HIRSH, H. (Ed.). *Proceedings of the Eleventh International Conference on Machine Learning*. [S.l.]: Morgan Kaufmann, 1994.

SCHIRMER, B. *Language and literacy development in children who are deaf*. New York, NY: Macmillan Publishing Co, 1994.

SCHUMEYER, R.; HEREDIA, E.; BARNER, K. Region of interest priority coding for sign language videoconferencing. In: *IEEE First Workshop on Multimedia Signal Processing*. Princeton: [s.n.], 1997. p. 531–536. Disponível em: <citeseer.nj.nec.com/schumeyer97region.html>.

SHUTT, J. *Recursive Adaptable Grammars*. 1993.

SHUTT, J. N. *Self-Modifying Finite Automata - Power and Limitations*. Worcester, Massachusetts, 1995.

SIPPU, S.; SOISALON-SOININEN, E. A syntax-error-handling technique and its experimental analysis. *ACM Transactions on Programing Language and Systems*, v. 5, n. 4, p. 656–679, 1983.

SMITH, M. R. G. D. E. Ordering conjunctive queries. *Artificial Intelligence*, v. 26, p. 171–215, 1985.

STENGER, B.; MENDONçA, P. S.; CIPOLLA, R. Model-based hand tracking using an unscented kalman filter. In: *Proc. British Machine Vision Conference*. Manchester, UK: [s.n.], 2001. I, p. 63–72. Disponível em: <citeseer.nj.nec.com/stenger01modelbased.html>.

STERLING, E. S. L. *The Art of Prolog: Advanced Programming Techniques*. [S.l.]: MIT Press, 1994.

STIEFELHAGEN, R.; YANG, J.; WAIBEL, A. Tracking eyes and monitoring eye gaze. *Proceedings of the Workshop on Perceptual User Interfaces (PUI'97).*, p. 98–100, 1997. Disponível em: <citeseer.nj.nec.com/stiefelhagen97tracking.html>.

TANIWAKI, C. Y. O.; NETO, J. J. *Autômatos Adaptativos no Tratamento Sintático de Linguagem Natural*. São Paulo, Brasil, 2001.

TRIESCH, J.; MALSBURG, C. von der. Robust classifications of hand postures against complex backgrounds. In: *Proc. of the 2nd Int. Conf. on Automatic Face and Gesture Recognition*. Killington, Vermont: [s.n.], 1996.

UTGOFF, N. C. B. P. E.; CLOUSE, J. A. Decision tree induction based on efficient tree restructuring. *Machine Learning*, v. 29, n. 1, p. 5–44, October 1997.

VILARES, M.; DARRIBA, V. M.; RIBADAS, F. J. Regional least-cost error repair. *Lecture Notes in Computer Science*, v. 2088, p. 293–301, 2001.

WANG, J. G.; SUNG, E. Gaze determination via images of irises. *Image and Vision Computing*, Elsevier-Science, v. 19, n. 12, p. 891–911, October 2001.

WARE, C.; MIKAELIAN, H. An evaluation of an eye tracker as a device for computer input. In: ____. [S.l.]: Elsevier, 1987.

WEGBREIT, B. *Studies in Extensible Programming Languages*. Cambridge, Massachusetts, May 1970.

WEGBREIT, B. *Extensible programming languages*. New York: Garland Publishing Inc., 1980.

WIJNGAARDEN, A. van. Revised report on the algorithmic language algol 68. *Acta Informatica*, v. 5, p. 1–236, 1974.

WITTEN, I. H.; FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. [S.l.]: Morgan Kaufmann, 2000.

WOLL, B.; SUTTON-SPENCE, R.; ELTON, F. The sociolinguistics of sign languages. In: ____. Cambridge, UK: Cambridge University Press, 2001. cap. Multilingualism - the global approach to sign languages.

YANG, Y.; WEBB, G. Proportional k-interval discretization for naive-bayes classifiers. In: *12th European Conference on Machine Learning - ECML01*. Freiburg, Germany: Springer-Verlag, 2001. p. 564–575.

YU, B. Coding and model selection: A brief tutorial on the principle of minimum description length. *Statistical Computing and Statistical Graphics Newsletter*, v. 9, n. 2, p. 27–32, 1998.