Gabriel Ambrósio Archanjo

Indução de Programas Genéticos Lineares para Modelagem de Processos de Manipulação de Informação

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientador: Fernando José Von Zuben

Campinas, SP
2012

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Archanjo, Gabriel Ambrósio
Ar22i Indução de programas genéticos lineares para modelagem de processos de manipulação de informação / Gabriel Ambrósio Archanjo. –Campinas, SP: [s.n.], 2012.

Orientador: Fernando José Von Zuben.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Programação genética. 2. Modelagem de processos. 3. Tecnologia da informação. I. Von Zuben, Fernando José. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação III. Título.

Título em Inglês: Induction of linear genetic programs for modeling data manipulation processes
Palavras-chave em Inglês: Genetic programming, Process modeling, Information technology
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca Examinadora: Gisele Lobo Pappa, Mario Jino
Data da defesa: 24/01/2012
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Gabriel Ambrósio Archanjo

Data da Defesa: 24 de janeiro de 2012

Título da Tese: "Indução de Programas Genéticos Lineares para Modelagem de Processos de Manipulação de Informação"

Prof. Dr. Fernando José Von Zuben (Presidente): Fernando José Von Zuben

Profa. Dra. Gisele Lobo Pappa: GLPappa

Prof. Dr. Mario Jino: Mario Jino

Resumo

Reproduzindo tendências verificadas em outros setores produtivos, métodos para automatizar etapas e reduzir custos têm sido propostos na área de desenvolvimento de software. Entretanto, a etapa mais trabalhosa, a codificação da solução, continua sendo realizada quase que exclusivamente por programadores humanos. Trabalhos na área de geração automática de programas para manipulação de dados têm focado predominantemente na descoberta de conhecimento e extração de padrões de bases de dados estáticas. Porém, para a modelagem de processos que normalmente alteram registros armazenados em bancos de dados, é necessário tratar os dados como entidades dinâmicas. Este trabalho apresenta uma abordagem para indução de programas via programação genética linear. Em termos de funcionalidade, os programas obtidos são capazes de consultar, inserir, excluir e atualizar registros num banco de dados relacional. O intuito é modelar processos de manipulação de informação, presentes em sistemas de tecnologia de informação. Os resultados indicam que a abordagem é capaz de implementar processos simples, gerando programas de computador consistentes e com interpretabilidade comparável à de programas escritos em linguagens de programação tradicionais.

Palavras-chave: Programação genética, Modelagem de processos, Tecnologia da informação.

Abstract

Reproducing trends observed in other productive branches, methods to automate stages and reduce costs have been proposed for software development. However, perhaps the most laborious step, the computer programming, is generally performed entirely by human programmers. Works in the field of automated generation of computer programs for data manipulation have been focused almost exclusively on knowledge discovery and pattern extraction in static datasets. Nevertheless, in the case of modeling processes that usually alter objects stored in databases, it is necessary to handle the dataset as dynamic entities. This work proposes an approach for program induction based on linear genetic programming. In terms of functionality, the obtained programs are able to query, delete, insert and update records stored in a relational database. The aim is to model processes for data manipulation, present in information technology systems. The results indicate that the proposed approach can implement simple processes, generating consistent programs as interpretable as the ones written in traditional programming languages.

Keywords: Genetic Programming, Process modeling, Information technology.

Agradecimentos

Ao meu orientador, Prof. Fernando José Von Zuben, pelos ensinamentos e pela confiança depositada em mim para execução deste projeto de pesquisa.

Aos colegas do Laboratório de Bioinformática e Computação Bio-inspirada (LBiC), pelas discussões calorosas e inspiradoras.

À Multiway, por valorizar a formação profissional e permitir a dedicação parcial a este projeto.

À minha namorada, Lorena, pela paciência e incentivo durante esta jornada.

Aos docentes e funcionários da Faculdade de Engenharia Elétrica e de Computação (Unicamp), pela estrutura necessária para a realização deste trabalho.

À CAPES, pelo apoio financeiro durante parte deste projeto.

À minha família.

Sumário

Lista de Figuras **xiii**

Lista de Tabelas **xv**

Lista de Símbolos **xvii**

Trabalhos Publicados Pelo Autor **xix**

1 Introdução **1**
1.1 Objetivos 3
1.2 Resultados Obtidos 3
1.3 Organização do Texto 4

2 Evolução do Processo de Desenvolvimento de Software **7**
2.1 Metodologias de Desenvolvimento de Software 8
2.1.1 Modelo em Cascata 9
2.1.2 Prototipação 10
2.1.3 Modelo em Espiral 12
2.1.4 Metodologias Ágeis e Iterativas 12
2.2 Princípios e Ferramentas Ágeis para Desenvolvimento de Software 14
2.2.1 Mapeamento objeto-relacional 14
2.2.2 Ferramentas de Teste Automatizadas 15
2.2.3 Frameworks 16
2.2.4 Padrão de Projeto de Software 17
2.3 Engenharia de Software Baseada em Busca 18
2.3.1 Estimação de Custo de Software 19
2.3.2 Teste de Software 20
2.3.3 Outras Aplicações 21
2.4 Considerações Finais 22

3 Programação Genética **23**
3.1 Representação dos programas 25
3.1.1 Representação em Árvore 26
3.1.2 Representação Linear 27
3.1.3 Representação em Grafo 28

3.2 Evolução de Programas 29
3.2.1 Inicialização 30
3.2.2 Avaliação 31
3.2.3 Seleção 32
3.2.4 Reprodução 32
3.2.5 Detecção de Introns 36
3.3 Programação Genética Fortemente Tipada 36
3.4 Considerações Finais 37

4 Aplicações de Programação Genética 39
4.1 Síntese de Circuitos Elétricos 40
4.2 Robótica 41
4.2.1 Navegação Autônoma 42
4.2.2 Robocup 43
4.3 Descoberta de Conhecimento 43
4.3.1 Utilização de Programação Genética 48
4.4 Mineração de Dados 49
4.5 Consultas em Bancos de Dados 50
4.5.1 *MASSON* 50
4.5.2 Bancos de Dados Relacionais 52
4.6 Considerações Finais 54

5 Modelagem de Processos de Manipulação de Informação 55
5.1 A Ferramenta LGPDB 57
5.1.1 Sistema de Gerenciamento de Banco de Dados 59
5.1.2 Módulo de Indução de Programas 61
5.2 Sistema de Gerenciamento de Biblioteca 72
5.2.1 Modelagem do Sistema 72
5.2.2 Definição do Banco de Dados 73
5.2.3 Escolha das funcionalidades 74
5.2.4 Elaboração dos Casos de Avaliação 74
5.2.5 Indução dos Programas de Computador 76
5.3 Propriedades de Banco de Dados de Indução 84
5.4 Melhorias e Experimentos Preliminares 88
5.4.1 Combinação de Regras 89
5.4.2 Composição de soluções utilizando múltiplos programas 89
5.5 Considerações Finais 91

6 Conclusão 95

Referências bibliográficas 98

Lista de Figuras

2.1 Fluxograma das etapas do modelo em cascata. Nesta metodologia, as fases são executadas de forma sequencial, da engenharia de sistemas à manutenção. 9
2.2 Diagrama das etapas que compõem a metodologia de prototipação. Partindo da coleta e refinamento dos requisitos, um protótipo pode ser trabalhado em múltiplos ciclos até a engenharia do produto. Adaptado de Pressman (2010). 11
2.3 Diagrama da metodologia modelo em espiral. Partindo do centro da espiral, etapas de prototipação, análise, modelagem e validação são repetidas em diversos ciclos, desenvolvendo o produto iterativamente. Adaptado de Boehm (1988). 13

3.1 Exemplo de programa representado em árvore que modela a função $f(x) = x^3 + x^2 + 4x$. 26
3.2 Exemplo de programa em representação linear que modela a função $f(x) = x^3 + x^2 + 4x$. 28
3.3 Exemplo de programa representado em grafo que modela a função $f(x) = x^3 + x^2 + 4x$. 29
3.4 Fluxograma do algoritmo de programação genética. 30
3.5 Operador de recombinação para representação em árvore. 34
3.6 Operador de recombinação para representação linear. 34

4.1 Circuito embrionário utilizado para a síntese de circuitos elétricos (Koza, Bennett III, Andre and Keane, 1996). 41
4.2 Simulador oficial da competição *Robocup* 44
4.3 Etapas do processo de descoberta de conhecimento, adaptado de Han & Kamber (2006). 45
4.4 Arquitetura da ferramenta *MASSON*, adaptado de Ryu & Eick (1996a). 50
4.5 Esquema do banco de dados orientado a objetos, adaptado de Ryu & Eick (1996a). . 52

5.1 Elementos que compõem a arquitetura da ferramenta LGPDB. 59
5.2 Exemplo de programa no formato de LGPDB. Em negrito estão destacadas instruções que são introns. 68
5.3 Fluxograma do algoritmo de evolução de programas do LGPDB. 71
5.4 Diagrama entidade-relacionamento do sistema simplificado de gestão de biblioteca. . 73
5.5 Estágios de evolução de um programa para prover a funcionalidade de consulta **Q4**. . 81
5.6 Estágios de evolução de um programa para prover a funcionalidade de exclusão **D3**. . 82
5.7 Estágios de evolução de um programa para prover a funcionalidade de inserção **I1**. . 83
5.8 Estágios de evolução de um programa para prover a funcionalidade de atualização **U1**. 84
5.9 Gráfico dos valores da função de adaptação por geração, relativos às funcionalidades **Q4**, **D3**, **I1**, **U1**. Os pontos em vermelho indicam as gerações em que ocorreu variação do valor da função de adaptação. 85

5.10 Grafo das associações de registros de tabelas no banco de dados. Os nós são registros de uma dada tabela e arestas representam relacionamentos, criados por meio de chaves. Em verde, são mostrados os registros reais, obtidos de bases de livros e artigos disponíveis na Internet. Em vermelho, são mostrados registros hipotéticos e com algumas desassociações. 87
5.11 O impacto causado no processo evolucionário por registros com desassociações. . . . 88
5.12 Programa induzido para modelar a funcionalidade **F1**, resultante da concatenação de quatro subprogramas induzidos separadamente. 92

Lista de Tabelas

2.1 Atributos da base de dados de projetos de software. 19

5.1 Operações suportadas pelo módulo SGBD. 61
5.2 Tipos de variáveis manipuladas por instruções de programa do LGPDB. 64
5.3 Tabelas e atributos do banco de dados relacional utilizado no experimento. 74
5.4 Funcionalidades do sistema simplificado para gestão de biblioteca que serão providas por programas gerados automaticamente. 75
5.5 Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **Q4**. 76
5.6 Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **D3**. 76
5.7 Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **I1**. 76
5.8 Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **U1**. 77
5.9 Resultado da indução de programas para prover um subconjunto de funcionalidades de um sistema de gerenciamento de biblioteca. 78
5.10 Subconjunto de tabelas de um sistema de gestão de operações financeiras. Atributos em itálico são chaves para relacionar tabelas. 90
5.11 Exemplo de caso de avaliação para a funcionalidade **F1** 91

Índice de Acrônimos

ADF	*Automatically Defined Functions*
API	*Application Programming Interface*
CGPS	*Compiling Genetic Programming System*
DW	*Data warehouse*
FSGP	*Function Sequence Genetic Programming*
GP	*Genetic Programming*
KDD	*Knowledge Discovery and Data mining*
LGP	*Linear Genetic Programming*
LGPDB	*Linear Genetic Programming for Databases*
MVC	*Model-view-controller*
OCR	*Optical Character Recognition*
ORM	*Object-relational Mapping*
PADO	*Parallel Algorithm Discovery and Orchestration*
SBSE	*Search-based Software Engineering*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
STGP	*Strongly Typed Genetic Programming*
TI	Tecnologia da Informação

Trabalho Publicado Pelo Autor

1. G.A. Archanjo, F.J. Von Zuben. “Induction of Linear Genetic Programs for Relational Database Manipulation”. *Proceedings of the 2011 IEEE International Conference on Information Reuse and Integration* (IRI’2011), Las Vegas, Nevada, EUA, pg. 347-352, Agosto 2011.

Capítulo 1

Introdução

Atualmente, em praticamente todos os campos de atuação da sociedade, processos são geridos com o auxílio de programas de computador. Apesar deste papel importante, a popularização da informatização de processos decisórios e produtivos é um fenômeno recente e ainda era tendência no início dos anos de 1990 (Main, 1990; Gurbaxani and Whang, 1991). Antes disso, a informatização era algo exclusivo de alguns nichos de atuação e de empresas com capacidade de realizar tal investimento. A adoção gradativa, com maior acentuação nos últimos 20 anos, por sistemas de tecnologia da informação (TI), demandou a criação e o aprimoramento de metodologias e ferramentas para desenvolvimento de software. Além da demanda, também cresceu a complexidade dos sistemas, devido à evolução das arquiteturas de computador e à integração de sistemas e serviços, incentivados principalmente pelo surgimento da Internet. Ao longo das últimas quatro décadas, as metodologias de desenvolvimento de software sofreram fortes mudanças, partindo de metodologias inspiradas em outros setores produtivos, como é o caso do modelo em cascata (Royce, 1970), a métodos mais ágeis de desenvolvimento (Larman, 2004), concebidos exclusivamente para lidar com o ambiente extremamente dinâmico que cerca o desenvolvimento de software. Além disso, ferramentas e princípios foram elaborados para dar suporte às inevitáveis mudanças que ocorrem em projetos de software ao longo do seu desenvolvimento.

Assim como em qualquer outro setor produtivo, métodos para automatizar etapas e reduzir custos têm sido propostos para a área de desenvolvimento de software. Ferramentas para automatização de testes de unidade, refatoração de código, geração automática de código a partir de diagramas, dentre outras, têm sido propostas. Além de ferramentas de automatização de processos, foram também propostas ferramentas de auxílio à tomada de decisão de processos de desenvolvimento de software. A área de engenharia de software baseada em busca (SBSE, do inglês *Search-based Software Engineering*) (Harman and Jones, 2001; Harman, 2007) tem explorado a aplicação de técnicas de otimização matemática a problemas relacionados a diversas etapas de processos de engenharia de software. Po-

rém, dentre todas as etapas presentes na concepção de um produto de software, a elaboração de algoritmos e codificação provavelmente é a mais trabalhosa e com menor grau de automatização. Portanto, a geração automática de programas, módulos computacionais ou algoritmos é um campo de pesquisa com potencial de gerar soluções que podem agregar valor a processos de desenvolvimento de software.

Existem técnicas para explorar a geração automática de programas de computador em vários níveis. No caso de programação automática (Balzer, 1985), programas são gerados a partir de uma outra representação que modela a solução para um dado problema. Portanto, neste caso, os elementos que resolvem o problema já estão previamente especificados num modelo de representação de alto nível. Sendo assim, neste caso, o desafio é converter a representação deste modelo em código de computador. Com menos restrições, programação evolucionária (Fogel et al., 1966) modela soluções computacionais através da otimização de parâmetros de uma representação de estrutura fixa. Neste caso, a estrutura das soluções candidatas está previamente configurada para tratar um determinado tipo de problema. Os parâmetros serão otimizados para atender diferentes cenários de um mesmo problema. Finalmente, com a programação genética (Koza, 1992), soluções são otimizadas tanto em nível de estrutura como de parâmetros. Nesta abordagem, o modelo de representação é muito semelhante a programas de computador, compostos por instruções que manipulam dados em memória, condicionais e laços. A flexibilidade do modelo implica no tamanho do espaço de busca de soluções. Portanto, quanto mais flexível o modelo, mais custoso e desafiador tende a ser o processo evolucionário. Sendo assim, a escolha da abordagem e forma pela qual soluções candidatas serão representadas vai depender do problema que será tratado.

A geração automática de programas para modelar processos de manipulação de informação é um problema desafiador. Ela difere de problemas de descoberta de conhecimento e mineração de dados, em que o banco de dados é utilizado para organizar e consultar informação. Além de organizar e consultar informação, os registros no banco de dados podem ser alterados no caso da modelagem deste tipo de processo. No caso de um sistema de gerenciamento de biblioteca, por exemplo, a modelagem do processo que cria um novo empréstimo entre um usuário e um livro precisa adicionar um novo registro no banco de dados. A forma pela qual este registro será adicionado está relacionada com a modelagem do sistema e os parâmetros do processo, neste caso uma identificação do usuário e outra do livro. Portanto, para este propósito, programas gerados automaticamente deverão consultar, excluir, inserir e atualizar dados no banco de dados. Além disso, a utilização de um banco de dados relacional - formato mais comum de organização de informação em sistemas de TI - aumenta o desafio, uma vez que, nesta representação, a informação é organizada de forma segmentada, com dados distribuídos em múltiplas tabelas relacionadas. Sendo assim, além da realização de filtros para obter o conjunto correto de registros que devem ser manipulados, programas gerados automaticamente

também deverão relacionar as tabelas corretamente para associar registros de tabelas distintas. Dado este cenário, é difícil imaginar uma representação com estrutura fixa, capaz de modelar soluções de diferentes complexidades, envolvendo múltiplas tabelas e formas distintas de manipulação de dados. Portanto, justifica-se o emprego de programação genética para este propósito.

1.1 Objetivos

O propósito geral deste trabalho é desenvolver um método de geração automática de programas de computador para modelagem de processos de manipulação de informação. Sendo assim, os objetivos específicos deste trabalho são:

1. Avaliar a evolução e o estado atual de processos de desenvolvimento de software, com enfoque em sistemas de TI, no intuito de identificar a oportunidade de atuação de uma ferramenta para geração automática de programas, que incorpora programação genética.

2. Analisar as diferentes propostas e trabalhos de programação genética para definir a representação do problema e o processo de geração automática de programas.

3. Introduzir uma nova abordagem para a geração automática de programas capaz de modelar processos de manipulação de informação.

Os objetivos estão relacionados, sendo que os dois primeiros são premissas para alcançar o último. A execução deste trabalho foi organizada de forma a alcançar os objetivos na ordem apresentada, para que, na etapa de elaboração de uma nova abordagem, todas as análises e conceitos necessários já estejam disponíveis para embasar as decisões.

1.2 Resultados Obtidos

A demanda crescente por métodos automáticos de geração de programas para a implementação de sistemas de TI foi a principal motivação deste trabalho. Portanto, do ponto de vista analítico, identificaram-se as tarefas do processo de desenvolvimento de software que se mostram mais indicadas para automatização e em quais tipos de tarefa a geração automática de programas já proporciona resultados expressivos.

Uma vez que a geração automática de programas, com capacidade de consultar, excluir, inserir e atualizar dados num banco de dados, é uma área pouco explorada, este trabalho apresenta um esforço inicial visando o emprego desta técnica para modelagem de processos de manipulação de informação. Sendo assim, diversos desafios permearam este projeto, com destaque para:

- Como deve ser o ambiente de indução de programas? Quais instruções e estruturas de dados em memória devem estar disponíveis aos programas genéticos?
- Quais métricas deverão ser utilizadas para determinar a distância entre a saída produzida por um programa candidato e o resultado desejado para as diferentes formas de manipulação de dados?
- Uma vez que programas genéticos precisam relacionar registros de diferentes tabelas corretamente para ter acesso a atributos discriminantes, como diferenciar, nas primeiras gerações do processo evolucionário, graus de aptidão de soluções candidatas que ainda não têm acesso a tais atributos?

Como resultado deste trabalho, obteve-se uma ferramenta capaz de gerar programas, numa linguagem interpretável por humanos, para modelar um determinado conjunto de funcionalidades de manipulação de informação em bancos de dados relacionais. Mesmo que estas funcionalidades sejam de fácil implementação para programadores humanos, a automatização permite que estes programadores se ocupem de tarefas de implementação mais complexas, que ainda não foram automatizadas.

Em suma, foi possível analisar a viabilidade de um ambiente de desenvolvimento de sistemas de TI que combina soluções implementadas por programadores humanos e soluções geradas automaticamente.

1.3 Organização do Texto

O Capítulo 2 avalia a evolução do processo de desenvolvimento de software, dando destaque à evolução das metodologias e ferramentas de suporte ao desenvolvimento. Este capítulo também analisa a utilização de ferramentas de auxílio à tomada de decisão em processos de desenvolvimento de software. O Capítulo 3 apresenta uma revisão da área de programação genética, abordando diferentes propostas e aspectos de implementação de abordagens que utilizam estes conceitos. Dada a flexibilidade e as variações de ambientes para indução de programas genéticos, o Capítulo 4 apresenta diversas aplicações de programação genética com enfoque nos diferentes ambientes desenvolvidos para abordar uma grande gama de problemas. A análise de abordagens alternativas apresentadas na literatura é utilizada para auxiliar a definição da abordagem apresentada neste trabalho. Finalmente, o Capítulo 5 apresenta a contribuição deste trabalho, uma ferramenta para geração automática de programas capazes de modelar processos de manipulação de informação. O Capítulo 6 apresenta uma análise dos resultados obtidos neste trabalho e perspectivas futuras de atuação.

Este texto foi organizado de forma a alcançar, de maneira sequencial, os objetivos apresentados na seção 1.1. Apesar dos objetivos estarem organizados seguindo um cronograma de projeto, sendo que

os dois primeiros são premissas necessárias para o último, não é obrigatória a leitura deste trabalho de forma linear. Seções que utilizam conceitos apresentados em outros capítulos realizam a devida menção para auxiliar o leitor que não está lendo este trabalho de forma linear. Mesmo assim, ao leitor familiarizado com engenharia de software e com a intenção de não ler o Capítulo 2, é recomendada a leitura da seção 2.3, presente neste último capítulo, pois aborda um assunto pouco explorado na literatura específica de engenharia de software: a utilização de métodos de otimização matemática para o auxílio à tomada de decisão em processos de engenharia de software. Da mesma forma, ao leitor familiarizado com programação genética, é recomendada a leitura do Capítulo 4, visto que os trabalhos selecionados e o enfoque das análises são preparatórios para o desenvolvimento da abordagem que representa a principal contribuição deste trabalho.

Capítulo 2

Evolução do Processo de Desenvolvimento de Software

Programas de computador se tornaram a principal ferramenta para gerenciamento de informações e processos da sociedade moderna. Atualmente, em praticamente todos os campos de atuação da sociedade, processos são geridos com o auxílio de programas de computador. Apesar deste papel importante, a popularização da informatização de processos decisórios ou produtivos é um fenômeno recente, passando a se estabelecer a partir dos anos 1990 (Main, 1990; Gurbaxani and Whang, 1991). Mais recentemente, além dos tradicionais computadores pessoais e de trabalho, outros diversos dispositivos móveis, com a capacidade de processar informação, surgiram ao longo dos anos. Graças à miniaturização do hardware de arquiteturas de computador, um dos nichos que mais cresceu foi o dos dispositivos móveis, com maior destaque para celulares inteligentes (*smartphones*), *tablets* e aparelhos de GPS. Sendo assim, na atualidade, além de organizarem e manipularem praticamente todos os processos por meio de software, as pessoas estão utilizando uma gama cada vez maior de dispositivos que demandam a necessidade de soluções integráveis, capazes de interagir com diversos aparelhos. No caso de sistemas bancários, por exemplo, operações financeiras podem ser feitas a partir de computador pessoal, caixa eletrônico, telefone via comando de voz e *smartphone* e *tablets* via aplicativos. Dado esse cenário extremamente dinâmico e envolvendo diversas variáveis, continua sendo um desafio garantir a integridade de todos os processos e a compatibilidade com todos os dispositivos, minimizar custos no caso de redundância de funcionalidades e sincronizar o desenvolvimento de soluções com diversos software envolvidos.

A Internet é outro elemento que influencia significantemente o desenvolvimento de software. Os benefícios relacionados a manutenção, segurança, compatibilidade, integrabilidade de software acessados por um navegador através da Internet estão promovendo uma mudança de paradigma. Cada vez mais utiliza-se a Internet como uma plataforma para realizar diversas atividades. Atualmente, estão

disponíveis diversas aplicações sofisticadas que podem ser acessadas diretamente do navegador, em qualquer lugar e numa diferente gama de dispositivos, desde software para preparar apresentações a editores de imagens. Mesmo no caso de aplicações executadas localmente nos dispositivos, a Internet pode servir de ponte de acesso a outros dispositivos e serviços *Web*.

Independentemente da plataforma utilizada para a execução de programas de computador e do grau de conectividade da aplicação, os projetos de software exigem metodologias sofisticadas para a gestão do projeto e continua sendo um desafio entregar programas de computador de qualidade, sem defeitos de fabricação, dentro do tempo e custos estimados. Comparadas com metodologias de desenvolvimento de produtos utilizadas em outros setores produtivos, as metodologias aplicadas a software são as mais recentes e menos amadurecidas. Além das metodologias, um conjunto de ferramentas de auxílio ao projeto de software continua em constante desenvolvimento, visando aumentar a qualidade de todas as etapas envolvidas.

O crescimento no número de dispositivos, a evolução da capacidade de processamento e memória de computadores e o aumento da complexidade dos sistemas demandaram melhores metodologias. As primeiras que foram formalizadas no início dos anos 1970 eram inspiradas em metodologias de outros setores produtivos. Até meados dos anos 1990, o objetivo dos pesquisadores na área de metodologias era melhorar a qualidade de soluções baseadas em software (Pressman, 2010). Finalmente, a partir dos anos 1990, foram popularizadas metodologias mais ágeis e capazes de tratar o ambiente extremamente dinâmico em que se encontram os projetos de software.

2.1 Metodologias de Desenvolvimento de Software

O desenvolvimento de produtos de software, assim como o desenvolvimento de qualquer outro tipo de produto, necessita de alguma metodologia que divida o processo de desenvolvimento em etapas, desde a concepção à entrega do produto final. Em linhas gerais, as metodologias para criação de produtos têm quatro etapas fundamentais: concepção, implementação, validação e entrega. Estas etapas podem ser expandidas em diversas subetapas. A etapa de concepção, por exemplo, numa abordagem simplista, pode acomodar apenas o projeto de um novo produto em que os requisitos são levantados apenas utilizando a intuição da equipe. Por outro lado, num projeto mais elaborado e envolvendo um investimento financeiro maior, esta etapa pode envolver análise de mercado, pesquisas com clientes, análise de produtos concorrentes, dentre outras atividades.

Esta seção apresenta as principais metodologias para desenvolvimento de software que surgiram ao longo das últimas quatro décadas. Apesar de cada metodologia ter conceitos e etapas específicas, de alguma forma, todas atendem às etapas básicas de desenvolvimento de um produto.

2.1.1 Modelo em Cascata

Modelo em cascata, proposto por Royce (1970), é a metodologia mais antiga e que foi mais amplamente usada em engenharia de software (Pressman, 2010). Nesta metodologia, o processo de desenvolvimento de um projeto é dividido e ordenado em etapas sequenciais, por exemplo: engenharia de sistemas, projeto, implementação, teste e manutenção. A Figura 2.1 apresenta o fluxograma de execução de etapas sequenciais. Uma determinada etapa só é iniciada com a conclusão da etapa anterior. Em sua concepção original, foi suposto que, após a finalização de uma etapa, não haveria necessidade de modificar os elementos envolvidos na mesma, por exemplo, os requisitos levantados na etapa de engenharia de sistemas.

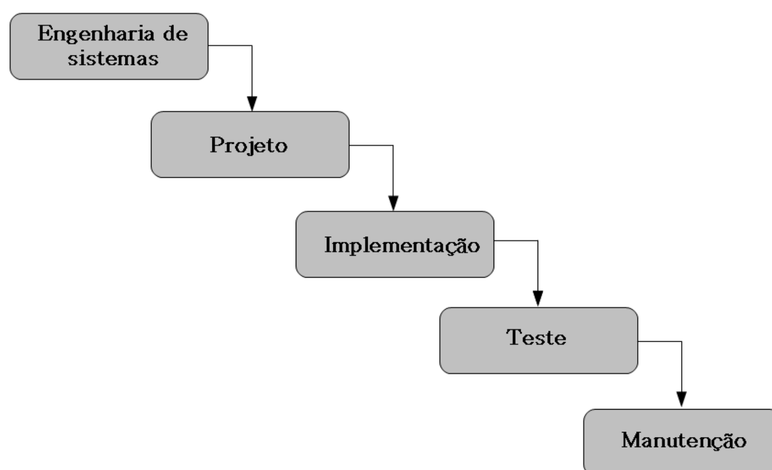

Fig. 2.1: Fluxograma das etapas do modelo em cascata. Nesta metodologia, as fases são executadas de forma sequencial, da engenharia de sistemas à manutenção.

Visto que este método de execução de projetos, com fases que vão da concepção à entrega do produto, tem sua origem na indústria de manufatura e construção civil, sua concepção considera restrições físicas em que modificações ao longo do projeto apresentam um custo elevado ou são infactíveis. Porém, a partir da década seguinte, o ambiente de desenvolvimento de software foi se mostrando cada vez mais dinâmico, necessitando múltiplas interações com clientes para se chegar numa definição do sistema. Parnas e Clements (1985) citam os seguintes fatores que dificultam uma definição detalhada de um projeto de software em sua fase inicial:

- Clientes ou usuários não têm certeza do que exatamente querem.
- Muitos detalhes que definem as necessidades de clientes ou usuários irão surgir apenas ao longo do processo de desenvolvimento.

- Clientes ou usuários são propensos a mudaram de ideia após interagirem com versões de desenvolvimento da aplicação.

- Elementos externos como, por exemplo, lançamento de aplicações semelhantes ou surgimento de novas tecnologias podem implicar mudanças no projeto.

O fato de que, no modelo em cascata, o cliente só terá um primeiro contato com o produto num estágio avançado do cronograma, em que modificações significativas implicam atrasar o projeto ou aumento significativo no custo, é a principal crítica em relação a esta metodologia. Apesar deste lado negativo, o modelo em cascata teve um papel importante na área de desenvolvimento de software, uma vez que formalizou e disseminou princípios organizacionais a serem aplicados a este tipo de projeto. As metodologias apresentadas nas próximas subseções tiraram proveito desta popularização e dos problemas identificados com o uso desta metodologia.

2.1.2 Prototipação

Com o objetivo de diminuir incertezas e validar conceitos já na fase inicial de desenvolvimento de um software, foi proposta a metodologia de prototipação. Conforme mencionado na seção anterior, uma das deficiências identificadas no modelo em cascata é o fato do cliente ter um primeiro contato com uma versão do software apenas num estágio avançado do projeto. Este contato tardio dificulta a execução de mudanças de grande impacto na definição do projeto. Afinal, este tipo de mudança precisa ser realizada na fase inicial de desenvolvimento, quando apenas uma pequena parte do software foi codificada.

Segundo Pressman (2010), a prototipação pode assumir três formas:

1. Um protótipo em papel ou software que retrata as interações homem-máquina.

2. Um protótipo funcional que implementa um subconjunto de funcionalidades do sistema.

3. Um software que implementa parte ou todas as funcionalidades, mas que será melhorado em novos esforços de desenvolvimento.

Independentemente da forma, fica claro que o objetivo da prototipação é tomar decisões de definição baseadas num modelo palpável, criado na fase inicial do projeto, capaz de validar os elementos que apresentavam um maior grau de incerteza. Na prototipação, é proposta uma sequência de etapas, desde a coleta e refinamento dos requisitos à engenharia do produto baseada no protótipo, conforme apresentado na Figura 2.2. Na primeira etapa, os requisitos são coletados e refinados com intuito de definir o software que será desenvolvido. Em seguida, na fase de "Projeto Rápido", é definido

qual o tipo de protótipo que será implementado e quais conceitos este deverá validar. Então o protótipo é implementado e avaliado pelo cliente. Provavelmente, mudanças conceituais e melhorias serão identificadas no protótipo, sendo feito um refinamento na fase seguinte. Se após esse refinamento o protótipo ainda não atender aos requisitos, volta-se à etapa de projeto rápido. Este ciclo pode ser repetido quantas vezes for necessário até a obtenção de um protótipo que atende a todos os requisitos, para finalmente entrar na etapa de engenharia de produto. Nesta última fase, o produto é modelado em maior detalhe utilizando um protótipo que minimiza as incertezas e valida os principais elementos relacionados ao projeto, aumentando assim a probabilidade de sucesso.

No desenvolvimento de qualquer tipo de produto, a concepção de algo tangível já na fase inicial é uma forma de avaliar conceitos que não estão claros, ou precisam ser tratados em maior profundidade, nos documentos de proposta de projeto. Porém, o desenvolvimento de protótipo por si só, não mapeia todos os elementos necessários no produto final. Sendo assim, é comum a utilização da prototipação como uma ferramenta para validação de conceitos em determinadas fases de uma metodologia maior, como é o caso do modelo em espiral, apresentado na seção seguinte.

Fig. 2.2: Diagrama das etapas que compõem a metodologia de prototipação. Partindo da coleta e refinamento dos requisitos, um protótipo pode ser trabalhado em múltiplos ciclos até a engenharia do produto. Adaptado de Pressman (2010).

2.1.3 Modelo em Espiral

A metodologia de prototipação, apresentada na seção anterior, mostra como diminuir incertezas existentes no projeto e aumentar a interação com o cliente por um processo de revisão, baseado no desenvolvimento incremental de um protótipo. Utilizando os conceitos de desenvolvimento incremental e maior interação com as pessoas responsáveis pelo produto final, a metodologia modelo em espiral (Boehm, 1988) expande esta ideia e incorpora outros diversos elementos necessários para gerir o desenvolvimento, desde o planejamento ao produto final. Esta metodologia, com diagrama ilustrado na Figura 2.3, é composta por atividades distribuídas em quatro conjuntos, separados pelos quadrantes da figura. A metodologia recebe esse nome, pois, partindo da elaboração do documento de conceito de operação, localizado no centro do diagrama, as outras etapas são realizadas seguindo uma espiral. Cada quadrante agrupa um conjunto de atividades para um propósito maior. No caso do quadrante superior direito, por exemplo, o objetivo das atividades é reduzir os riscos do projeto por meio de análises e desenvolvimento de protótipos. De forma iterativa, o produto é desenvolvido ao longo de múltiplos ciclos com planejamento, análise de riscos, prototipação e desenvolvimento.

Cada ciclo pode ser considerado um projeto menor que é revisado pelas pessoas que decidem como deve ser o resultado final do produto, após cada atividade de planejamento. Essa maior interatividade com os responsáveis pelo produto aumenta a probabilidade de sucesso do projeto. No pior caso, em que o produto é inviável, esta metodologia auxilia chegar a tal conclusão o quanto antes, economizando tempo e dinheiro.

2.1.4 Metodologias Ágeis e Iterativas

Comparando as metodologias modelo em cascata e espiral, fica claro que a dinamicidade do ambiente de desenvolvimento de software exigiu que as metodologias incorporassem elementos para aumentar a interatividade com as pessoas que tomavam as decisões de concepção da aplicação, por exemplo, clientes, usuários ou gerente de produtos, e elementos para diminuir as incertezas que permeiam o projeto, principalmente em sua fase inicial. Dentre a metade dos anos 1990 e o início dos anos 2000, diversas metodologias visando um processo de desenvolvimento mais ágil, iterativo e evolucionário foram propostas como, por exemplo, o *Scrum* (Sutherland, 1995), *Extreme Programming* (Beck, 2001), *Rational Unified Process* e *Test-Driven Development* (Beck, 2003). Um fato marcante que ocorreu nesta época foi a publicação de um manifesto (Beck et al., 2001) que define a abordagem ágil para desenvolvimento de software. Este manifesto promoveu o desenvolvimento e a divulgação de metodologias que incorporam os elementos desta abordagem. Apesar de cada metodologia ter suas características específicas, todas compartilham alguns princípios fundamentais para um desenvolvimento ágil e iterativo (Larman, 2004):

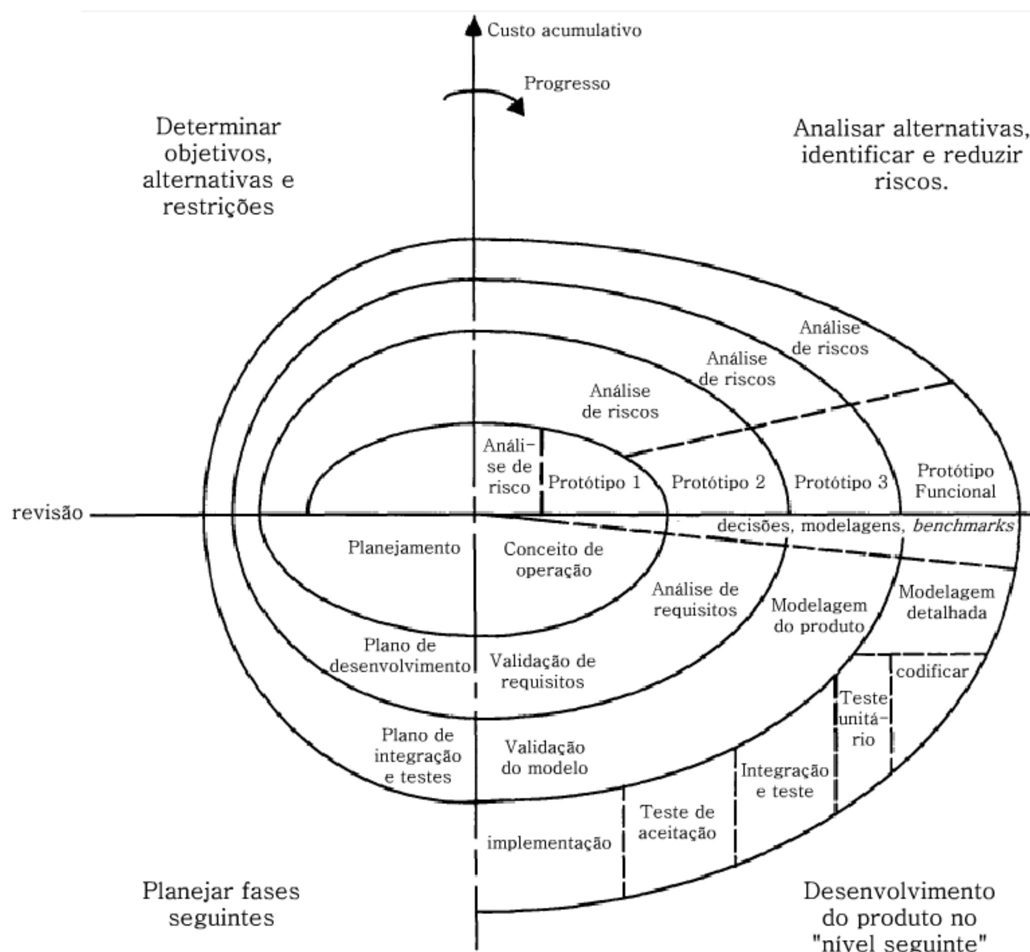

Fig. 2.3: Diagrama da metodologia modelo em espiral. Partindo do centro da espiral, etapas de prototipação, análise, modelagem e validação são repetidas em diversos ciclos, desenvolvendo o produto iterativamente. Adaptado de Boehm (1988).

- **Ciclos de projeto:** os projetos são quebrados em ciclos que são praticamente mini-projetos. Normalmente esses ciclos contemplam etapas que vão desde o planejamento ao teste do produto. Sendo assim, é comum que cada ciclo produza uma versão funcional do produto, testada e validada pelo cliente. O número de ciclos e de dias de cada ciclo pode variar com o projeto ou metodologia.

- **Interação com clientes:** as pessoas responsáveis por delinear as características do produto estão formalmente envolvidas no projeto e são consultadas a cada ciclo. Portanto, estas pessoas acompanham o desenvolvimento do produto continuamente, garantindo que todos os conceitos foram entendidos plenamente e o projeto está de acordo com os requisitos estabelecidos.

- **Mudanças são naturais:** ao invés de questionar os clientes, argumentando que eles estão

pedindo algo diferente do que foi combinado inicialmente ou estabelecido em contrato, estas metodologias entendem e incorporam as incertezas de projeto. Uma vez que clientes mudam de opinião ao longo do projeto, possivelmente influenciados por resultados parciais, produtos concorrentes, novas tendências, entre outros fatores, é melhor estar preparado para tratar esta questão do que tentar combatê-la. Na fase inicial, o projeto deve ser definido em linhas gerais, detalhes específicos serão definidos ao longo das iterações, principalmente porque o resultado parcial de um ciclo pode influenciar decisões nos ciclos seguintes.

- **Métrica baseada no produto de software:** analisar o progresso de um produto de software baseado apenas em documentos como, por exemplo, cronogramas, projeções ou imagens de telas, pode resultar num mau entendimento do estado atual do software. Este tipo de documento até pode auxiliar a análise, porém, seguindo o princípio de desenvolvimento ágil, a atual versão do produto de software deve ser o fator mais influente nesta análise. Usando o software resultante de alguns ciclos de desenvolvimento, pode-se determinar a quantidade de itens do projeto que já foram implementados (e.g., telas e módulos) e se os mesmos realmente atendem todas as especificações. Este tipo de análise pode evitar que partes do produto, que foram consideradas como finalizadas, tenham que ser modificadas no final do projeto, com maior custo por terem sido desenvolvidas com esforço reduzido, parcialmente ou com qualidade inferior à desejada para atender uma métrica de progresso, por exemplo, baseada em cronograma de módulos finalizados.

2.2 Princípios e Ferramentas Ágeis para Desenvolvimento de Software

Na seção anterior, foram mostrados a evolução na área de metodologias de desenvolvimento de software e quais elementos destas metodologias auxiliam no desenvolvimento deste tipo de produto. Entretanto, além de metodologias para definir etapas e gerenciar os processos de desenvolvimento, ao longo dos anos foram desenvolvidas ferramentas para melhorar o processo de desenvolvimento ou o próprio software, que acabaram sendo usadas por diversas metodologias. Nesta seção são apresentadas algumas ferramentas amplamente utilizadas que independem da metologia de desenvolvimento de software adotada.

2.2.1 Mapeamento objeto-relacional

Um problema comum em desenvolvimento de sistemas de TI é a manipulação de dados em duas representações, uma orientada a objetos e armazenada em memória, outra representada em forma de

tabelas e relações, armazenada num banco de dados relacional. Para evitar a necessidade de implementação manual de algoritmos para conversão de uma representação para a outra, foram criadas ferramentas de mapeamento objeto-relacional (ORM, da sigla em inglês *Object-relational Mapping*) que são capazes de realizar essa tarefa automaticamente, utilizando um arquivo que descreve o mapeamento.

Ambler (2000) descreve diferentes técnicas para realizar esse tipo de mapeamento, por exemplo, toda hierarquia de classes de objeto numa única tabela, cada classe de objeto numa tabela própria ou toda classe mapeada numa tabela de estrutura genérica. Independentemente da técnica de mapeamento, é possível automatizar a conversão entre os formatos e a implementação de métodos básicos de consulta, armazenamento, atualização e exclusão. Desta forma, é possível reduzir o número de linhas de código, custos de desenvolvimento e manutenção de uma aplicação deste gênero.

2.2.2 Ferramentas de Teste Automatizadas

Uma etapa fundamental no desenvolvimento de qualquer tipo de produto é a fase de testes. Esta etapa pode definir se um produto terá sucesso ou irá fracassar. A entrega de um produto de menor qualidade, contendo defeitos, pode implicar desde a perda de credibilidade num produto ou marca até colocar a vida de usuários em risco. Atualmente temos software atuando em diversos segmentos em que não se admite falhas, incluindo sistemas de gerenciamento de transações financeiras e controles de equipamentos eletrônicos num avião. Falhas em software desses gêneros podem causar consequências gravíssimas, tanto que algumas modalidades de software são auditadas e certificadas, visando minimizar as falhas o máximo possível.

No caso de teste de software, a estratégia pode ser separada em quatro fases (Pressman, 2010):

- **Teste de unidade:** foca cada módulo individualmente, garantindo que funcione adequadamente como uma unidade.
- **Teste de integração:** nesta fase, é testado o funcionamento dos módulos em conjunto, ou seja, se a integração foi realizada de forma correta.
- **Teste de validação:** verifica se a aplicação atende as especificações levantadas na fase de análise de requisitos.
- **Teste de sistema:** finalmente, a aplicação é testada no ambiente real, em que são considerados fatores como hardware, pessoas (usuários reais) e desempenho.

Entretanto, a etapa de teste não é aplicada apenas na concepção do produto. Toda modificação no sistema, seja esta uma melhoria ou a adição de uma nova funcionalidade, implica em testar novamente todos os módulos envolvidos. O custo adicional para realização de todos os testes necessários pode

chegar a 40% de todo o esforço despendido num projeto de software (Pressman, 2010). Atualmente, é difícil imaginar um sistema que opere durante muitos anos numa mesma versão. Esta inevitável evolução é consequência de diversos fatores, por exemplo, a dinamicidade dos processos que estes sistemas modelam, a evolução das tecnologias que influenciam o sistema e a constante demanda por novas funcionalidades, visando atender às necessidades de usuários.

Dado este cenário, ferramentas de teste automatizadas são uma alternativa para reduzir custos e aumentar a qualidade no desenvolvimento de sistemas. Dentre as diferentes estratégias de automação de rotinas de teste, a mais popular é o teste de unidade automatizado. Nesta metodologia, unidades do programa - menor parte testável - são testadas separadamente por meio de asserções que verificam se o programa se comporta da forma esperada durante a execução. Para cada módulo ou funcionalidade a ser testada, são criados casos de teste, usualmente contendo uma ação e um resultado esperado. As asserções verificam se a versão atual do módulo ou funcionalidade está de acordo com as especificações. Após uma modificação no sistema, utilizando uma ferramenta para este propósito, todas as unidades de programa são testadas automaticamente e um relatório pode ser gerado, informando quais unidades falharam no teste.

Este tipo de ferramenta é utilizada em diversas metodologias ágeis de desenvolvimento de software. Porém, no caso da metodologia *Test-driven Development* (TTD) (Beck, 2003), em que desenvolvedores, numa primeira etapa, implementam as rotinas de teste e, em seguida, implementam o software que será testado, esta ferramenta é o elemento principal da metodologia.

Alguns trabalhos na literatura analisaram o impacto que esta metodologia causa em projetos de software em grandes corporações. Williams et al. (2009) analisaram a impacto da adoção da metodologia TTD em projetos na *Microsoft*. A metodologia foi utilizada por um time com 32 desenvolvedores e os resultados mostraram melhora de qualidade na nova versão de um produto. Em Maximilien & Williams (2003), é descrita a experiência de utilização desta metodologia em projetos na *IBM*, também apontando resultados positivos como, por exemplo, a redução de taxa de defeitos em 50%. Por outro lado, em ambos os trabalhos, foi apontado um maior custo de desenvolvimento, uma vez que a equipe gastou uma parcela do tempo implementando os casos de teste. Entretanto, conforme apontado por Williams, considerando o desenvolvimento de casos de teste de forma iterativa ao longo de novas versões do produto, espera-se que a produtividade da equipe aumente, principalmente pelo fato de que uma implementação de software de maior qualidade diminui os custos de manutenção.

2.2.3 Frameworks

Como já mencionado, sistemas de TI estão presentes praticamente em todos os setores de atuação da sociedade. É inevitável que, considerando essa enorme quantidade de sistemas, existam funcionalidades semelhantes entre eles, fazendo com que engenheiros de software, em diferentes empresas,

trabalhem em soluções distintas para um mesmo problema. O primeiro passo para minimizar este problema foi a identificação de funcionalidades semelhantes em sistemas distintos como, por exemplo, manipulação de bancos de dados, geração de gráficos estatísticos, criptografia de dados e desenvolvimento de interfaces gráficas. Identificadas estas funcionalidades, aplicações semi-completas formadas por componentes customizáveis, conhecidas como *frameworks* (Fayad et al., 1999), foram implementadas. Esta solução permite que, na implementação de um novo sistema, a equipe foque em questões específicas. Em outras palavras, questões tecnológicas relacionadas à segurança da informação, acesso a banco de dados ou componentes de interface sofisticados serão resolvidas utilizando software de terceiros, na maioria dos casos, distribuídos como *frameworks*. Além do reaproveitamento de funcionalidades, houve um aumento significativo de qualidade de produtos de software que utilizam este tipo de solução, uma vez que empresas começaram a focar exclusivamente no desenvolvimento de *frameworks*, tornando-os cada vez mais sofisticados.

Desde o crescimento na demanda por software comerciais, *frameworks* para diversas aplicações tem sido desenvolvidos e atualmente existe um vasto catálogo de soluções para praticamente todos os campos de atuação em software, desde componentes de interface a reconhecimento óptico de caracteres (OCR, do inglês *Optical Character Recognition*). Visto que muitas empresas estão atuando exclusivamente no desenvolvimento deste tipo de solução, um novo nicho de negócio foi criado. Um fenômeno recente, conhecido como computação em nuvem, também tem ganhado importância na área de reaproveitamento de soluções. Dentre as aplicações, esta abordagem permite disponibilizar soluções em forma de serviços na Internet, utilizando, por exemplo, *Web Services*. Soluções em software com acesso à Internet podem utilizar soluções terceiras para fornecer determinada funcionalidade (Bichier and Lin, 2006), por exemplo, utilizar o *Google Maps API* para exibir um mapa ou consultar cotações de ações através do *Yahoo Finance Web Services*.

2.2.4 Padrão de Projeto de Software

Um dos desafios na área de engenharia de software é definir quais partes do sistema serão específicas para um determinado módulo, funcionalidade ou para o próprio sistema, e quais devem ser desenvolvidas de forma mais genérica para que possam ser reutilizadas, inclusive no desenvolvimento de outros sistemas. Linguagens de programação orientadas a objeto facilitam o desenvolvimento de aplicações modulares e com trechos reaproveitáveis de código. A subseção anterior mostrou uma forma de reutilizar algoritmos ou funcionalidades completas pela utilização de *frameworks* e serviços. Entretanto, o reaproveitamento de soluções pode ir além de implementações. É comum que engenheiros de software experientes já tenham um arcabouço de modelagens reaproveitáveis, comumente utilizadas em múltiplos processos.

Com o intuito de facilitar a reutilização de boas modelagens, validadas e implementadas em diver-

sos sistemas, por um maior número de profissionais responsáveis por desenvolvimento de sistemas e, principalmente, por profissionais menos experientes, foram definidos padrões de projeto de software para diversas funcionalidades (Gamma et al., 1995). Cada padrão de projeto descreve a solução para um problema recorrente no desenvolvimento de sistemas. Estas modelagens normalmente abordam questões de arquitetura de software e são pouco dependentes de tecnologia ou linguagem de programação. Optando por utilizá-los, engenheiros de software aproveitam soluções amadurecidas durante anos, evitam soluções que comprometem reusabilidade e melhoram a documentação e manutenção do sistema, uma vez que módulos que seguem padrões de projetos são de maior compreensão para outros profissionais. Atualmente é possível encontrar diversos catálogos de padrões de projeto, direcionados a uma grande gama de aplicação.

2.3 Engenharia de Software Baseada em Busca

Em praticamente todas as ciências, existem problemas em que o objetivo é escolher a melhor combinação de itens dentre um conjunto de opções e restrições. Na área de logística, por exemplo, um conjunto de entregas de mercadoria pode ser feito em diferentes ordens e através de diferentes rotas. Num primeiro cenário, pode ser interessante encontrar a combinação que tem o menor custo, considerando, por exemplo, um compromisso entre consumo de combustível e o custo para trafegar em cada estrada. Num outro cenário, deseja-se encontrar o caminho mais rápido, independentemente do custo. Problemas deste tipo podem ser representados matematicamente, tendo o objetivo em forma de uma função algébrica, com parâmetros (escolhas) e restrições, e pelo emprego de técnicas de otimização matemática é possível maximizar ou minimizar o valor desta função em busca da melhor escolha para um dado objetivo.

Engenharia de Software Baseada em Busca (SBSE, do inglês *Search-based Software Engineering*) (Harman and Jones, 2001) é uma abordagem para auxiliar a tomada de decisão na área de engenharia de software, pelo emprego de técnicas de otimização. Desta forma, para um problema nesta área, são definidas uma representação, contendo parâmetros ajustáveis, e um método de avaliação. Técnicas de otimização são aplicadas para sugestão de conjuntos de parâmetros, servindo de suporte a processos decisórios na área de engenharia de software. Esta técnica permite, por exemplo, realizar predições utilizando características de um projeto de software em desenvolvimento e uma base de dados contendo características de projetos que já foram implementados. A seguir, SBSE é analisada em maior profundidade, considerando duas aplicações, uma na área de estimação de custo e outra na área de teste de software.

2.3.1 Estimação de Custo de Software

Uma etapa fundamental de qualquer projeto de software é o planejamento em que esforço humano, duração e custo são estimados em função do tamanho do projeto (Pressman, 2010). Existem diversos atributos que podem ser utilizados para medir o tamanho de um software como, por exemplo, número de linhas, telas, módulos, transações e relatórios. Uma forma de obter uma estimativa de custo é relacionar esses atributos com a produtividade e custo da equipe. Para estimar o custo de desenvolvimento das telas, por exemplo, o número de linhas de código por tela é estimado e relacionado com a produtividade das pessoas envolvidas, medida em KLOC (milhares de linha de código) por pessoa mês, e o custo mensal de cada pessoa. Tanto as estimativas de número de linhas de código, como as de produtividade, devem ser baseadas num histórico de projetos realizados. Quanto maior a semelhança entre o projeto atual e um conjunto de projetos passados, mais precisas tendem a ser as estimativas.

Este tipo de problema, em que se deseja determinar a relação entre dois conjuntos de atributos, tem sido abordado extensivamente na área de mineração de dados para classificação de padrões, predição de valores e aproximação de funções, por meio de técnicas que incorporam otimização matemática. Sendo assim, a estimação de custo ou tamanho de software pode ser modelado com uma abordagem de SBSE. Em Burgess & Lefley (2001), foi empregada esta técnica para estimação de esforço de desenvolvimento de software, utilizando uma base de dados contendo atributos de 81 projetos de software. Neste caso, ao invés de utilizar o número de linhas de código como a menor unidade de software, foi utilizado o número de pontos de função, métrica proposta por Albrecht (Albrecht, 1979; Albrecht and Gaffney Jr, 1983), em que a menor unidade de software é uma funcionalidade. A vantagem deste método é que tem menor dependência com a tecnologia utilizada.

Tab. 2.1: Atributos da base de dados de projetos de software.

Atributo	**Descrição**
Project	Identificador numérico.
ExpEquip	experiência da equipe em anos.
ExpProjMan	experiência do gerente de projetos em anos.
Transactions	número de transações.
Entities	número de entidades.
RawFPs	pontos de função não ajustados.
AdjFPs	pontos de função ajustados
DevEnv	ambiente de desenvolvimento
YearFin	ano de finalização
Envergure	medida de complexidade
Effort	medido em hora por pessoa

Como mostrado na Tabela 2.1, a base de dados contém um atributo dependente, esforço (*effort*), e nove atributos independentes. O objetivo é induzir um modelo capaz de mapear a relação entre esses nove atributos e o esforço necessário para implementar um dado software. Para esta finalidade, a base de dados foi separada em dois conjuntos, um para treinamento (63 amostras) e outro para validação (18 amostras). De modo geral, as abordagens que incorporam um método de otimização, de modo iterativo, manipulam parâmetros que mapeiam variáveis independentes (entrada) e as variáveis dependentes (saída), utilizando as amostras de treinamento. Já as amostras de validação são utilizadas para avaliar o modelo utilizando relações de entrada e saída que não foram utilizadas na fase de treinamento. Para avaliar a qualidade do mapeamento, é utilizada uma função de avaliação que compara os valores sugeridos pelo modelo e os valores presentes na base de dados, relativos às amostras de validação. A partir desta diferença, pode-se considerar o erro, diferença entre esses valores, ou o índice de acerto, dado um intervalo de tolerância. Como já explicado anteriormente, no caso de um método de avaliação baseado no erro, o objetivo do algoritmo será minimizar o valor da função. Por outro lado, no caso da avaliação por índice de acerto, o algoritmo irá maximizar o valor da função.

Neste trabalho foram empregados os métodos regressão linear, KNN (*k-nearest neighbor*), rede neural artificial e programação genética. Estas abordagens utilizam diferentes conceitos para atender o mesmo objetivo: mapear a relação entre os 9 atributos independentes e o esforço necessário para implementação de um projeto de software. Como era de se esperar, considerando a extensa utilização dessas técnicas em problemas semelhantes de outras áreas, os resultados indicam a viabilidade de técnicas de SBSE para suporte à tomada de decisão na área de estimação de custo.

Um outro exemplo de aplicação equivalente é mostrado por Dolado (2000). Neste caso, ao invés de estimar o esforço, é estimado o número de módulos necessários para implementar um projeto de software, por meio do uso de regressão linear múltipla, rede neural artificial e programação genética. A partir do número de módulos, é possível realizar prospecções relativas a esforço e custo de implementação de um determinado projeto.

2.3.2 Teste de Software

Na Subseção 2.2.2 foi apresentado o conceito de automatização de teste unitário em que, automaticamente, todos os módulos de uma aplicação podem ser testados e os resultados sumarizados em um relatório. Porém, mesmo utilizando esta técnica, que minimiza significantemente a presença de defeitos em versões testadas da aplicação, não existe garantia de que os módulos testados não apresentem defeitos, principalmente por duas razões: não há garantia de que os casos de teste cubram todos os cenários possíveis e podem existir defeitos nos próprios casos de teste. Portanto, diferentes técnicas de teste de software podem ser combinadas para maximizar o resultado.

Em Boden & Martino (1996), foram empregados algoritmos genéticos para encontrar defeitos

em APIs com o desenvolvimento finalizado. Neste caso, do ponto de vista de otimização, o objetivo é encontrar a sequência de chamadas de métodos providos por APIs, incluindo os valores de seus parâmetros, que geram o maior número de erros ou exceções disparadas pelo sistema operacional. Para este propósito, foi criada uma população de indivíduos que têm em seu código genético uma sequência de funções de APIs e seus parâmetros. De forma iterativa, cada indivíduo é interpretado, ou seja, são realizadas as chamadas de métodos especificadas no código genético e o retorno é avaliado. Quanto maior o número de problemas causados por um indivíduo, melhor este é avaliado. Após a avaliação de toda a população, os melhores indivíduos são selecionados, reproduzidos e modificados, pelo emprego de operadores genéticos. Este processo é repetido e, ao longo das iterações, pelo princípio da seleção natural, sequências de chamadas de métodos que destacam defeitos nas APIs tendem a emergir no código genético das soluções candidatas, visto que este tipo de sequência aumenta a nota de avaliação dos indivíduos. Após um determinado número de gerações, o processo é interrompido e os melhores indivíduos são analisados no intuito de encontrar novos defeitos nas APIs. Por meio deste método, foi possível detectar defeitos que não tinham sido detectados com os métodos tradicionais. Os conceitos de computação evolutiva utilizados por algoritmos genéticos são apresentados em maior profundidade no Capítulo 3, que trata de programação genética, uma outra técnica que utiliza esses conceitos.

Também empregando algoritmos genéticos, porém num outro campo da área de teste de software, o trabalho de Brian et al. (2005) apresenta uma abordagem para tentar encontrar a melhor combinação de escalonamento de tarefas para realização de teste de estresse de software de execução em tempo real. Neste caso, o código genético dos indivíduos contém uma sequência de índices de tarefas que precisam ser executadas e o momento em que elas devem entrar em execução. Neste caso, do ponto de vista de otimização, o objetivo é encontrar a sequência de execução de tarefas que apresenta o maior atraso. Em outras palavras, deseja-se encontrar o pior cenário de execução para determinar a necessidade de realização de otimizações. Aplicando a mesma sequência de passos especificados no exemplo anterior, ao longo das iterações emergem cenários que provocam atrasos de execução cada vez maiores. Analisando os diferentes cenários apontados, é possível determinar a necessidade de otimizações para acomodar as tarefas de forma a maximizar a probabilidade de que todas serão executadas dentro do tempo determinado.

2.3.3 Outras Aplicações

As duas subseções anteriores mostraram a aplicação de SBSE com o intuito de auxiliar processos decisórios na área de engenharia de software. Porém, as aplicações não se restringem apenas a esses exemplos. Muitos trabalhos começaram a explorar o potencial de SBSE para outras atividades como, por exemplo, para engenharia de requisitos (Bagnall et al., 2001), manutenção automática (Harman

et al., 2002), otimização de compilador (Cooper et al., 1999), garantia de qualidade (Khoshgoftaar et al., 2004), dentre outras. Uma revisão da área de SBSE, contendo mais exemplos de aplicação e apresentando uma visão mais aprofundada, pode ser encontrada em Harman (2007).

2.4 Considerações Finais

Neste capítulo, foi apresentada a evolução das diferentes formas de gerir o processo de desenvolvimento de software. Um conceito que ficou claro, analisando a evolução ocorrida ao longo dos anos, foi o ganho de agilidade e interatividade nos processos de desenvolvimento, principalmente com o objetivo de validar ou reavaliar os principais pontos do projeto, em múltiplas etapas do processo de desenvolvimento, minimizando incertezas até a conclusão do produto de software.

O surgimento de técnicas e ferramentas específicas para tratar de problemas de desenvolvimento de software, independentes da metodologia de desenvolvimento, também se mostraram cruciais para um aumento de produtividade e qualidade dos processos. A utilização de *frameworks*, padrões de projeto e ferramentas para ORM, por exemplo, estão presentes na maioria das aplicações comerciais. Por outro lado, técnicas de SBSE ainda não conquistaram o mundo corporativo, sendo a maioria dos trabalhos e ferramentas ligados a projetos acadêmicos. Entretanto, considerando o potencial e a crescente utilização de técnicas de otimização na área de computação, é esperada uma maior adesão desta técnica em aplicações comerciais.

Conforme apresentado neste capítulo, a evolução de metodologias e o desenvolvimento de ferramentas propiciaram o aumento de produtividade ou até mesmo a automatização de determinadas etapas relacionadas ao desenvolvimento de software. Entretanto, talvez a etapa mais trabalhosa, a elaboração de algoritmos e codificação, continua sendo realizada quase que exclusivamente por programadores humanos. Nos próximos dois capítulos, esta questão será abordada com o intuito de fundamentar os principais conceitos necessários para apresentar uma nova abordagem para geração automática de programas, focada em processos de manipulação de informação comuns em sistemas de TI.

Capítulo 3

Programação Genética

Desde o surgimento dos primeiros computadores digitais, programas de computador têm sido desenvolvidos para automatização de processos e aquisição de conhecimento. No primeiro caso, o comportamento e os resultados de execução do programa são previsíveis, uma vez que este é composto por uma sequência de etapas, tendo todos os seus elementos conhecidos, com o objetivo de modelar uma tarefa trabalhosa, normalmente repetitiva. Programas deste tipo foram desenvolvidos, por exemplo, para automatizar o cálculo de folha de pagamento de milhares de funcionários. Por outro lado, no segundo caso, programas foram desenvolvidos para adquirir um novo conhecimento que requer a execução de uma sequência muito grande de passos, inviável de ser executada manualmente. Nesta área, se destacam, por exemplo, programas de simulação e algoritmos de busca. A possibilidade de desenvolver um programa que, quando executado, é capaz de produzir algo novo, logo levantou a questão: seria possível desenvolver programas capazes de gerar novos programas, assim automatizando o próprio desenvolvimento de programas? Neste caso, a geração automática de programas significa a indução de um algoritmo, ou seja, encontrar uma sequência de etapas que modela uma determinada solução para um dado problema. Portanto, este conceito é diferente de programação automática (Balzer, 1985), em que diversas técnicas são empregadas para gerar programas de computador a partir de uma outra representação em alto nível que já modela a solução para um dado problema. Neste último caso, o objetivo é converter um determinado modelo de solução em programa executável.

O campo de pesquisa que serviu de alicerce para o desenvolvimento de abordagens para a geração automática de programas é a computação evolucionária (CE). Este campo foca na modelagem de soluções computacionais na teoria da seleção natural, proposta por Charles Darwin (1859). Esta teoria descreve como mudanças em indivíduos, numa determinada população de organismos, emergem ao longo das gerações. Indivíduos mais adaptados ao meio ambiente têm maior probabilidade de sobreviver e, consequentemente, maior probabilidade de propagar suas características genéticas para as

futuras gerações. Como consequência da geração de novas sequências de genes, resultantes da combinação de códigos genéticos pelo processo de reprodução sexuada ou da adição de novos elementos por processo de mutação, variações de indivíduos são adicionadas à população de organismos. Pelo princípio da seleção natural, variações benéficas aos indivíduos tendem a se perpetuar nas futuras gerações, enquanto que as variações indesejáveis tendem à extinção.

O emprego deste conceito em aplicações computacionais teve início nos anos 1960 e ganhou popularidade com os trabalhos de Rechenberg (1973), na área de estratégias evolutivas, e Holland (1975), na área de algoritmos genéticos. Neste último, é definido um *framework* para representar e manipular soluções computacionais utilizando conceitos de genética e de teoria da evolução natural, adotado até hoje por algoritmos de CE. Inclusive, neste mesmo trabalho (Holland, 1975) (Capítulo 8), Holland discute a limitação de representações fixas, em que apenas os parâmetros são otimizáveis, e a possibilidade de evoluir a própria representação de soluções candidatas. Após uma década, Cramer (1985) induziu pequenas sequências de funções de computador para modelagem de operações matemáticas simples, com um valor de saída em função de dois valores de entrada. Finalmente, em 1992, Koza (1992) formalizou a programação genética (GP, do inglês *Genetic Programming*), *framework* que permite a evolução de programas com as mesmas características dos desenvolvidos por programadores humanos. Utilizando GP, é possível induzir programas com instruções condicionais, laços, chamadas de função e outros elementos encontrados em linguagens de programação imperativas utilizadas por programadores humanos. Inclusive, como mostrado por Nordin (1994), é possível utilizar representação de baixo nível para programas genéticos, podendo assim executá-los direto no processador.

Antes de comparar GP com outras abordagens, é interessante mencionar o teorema *No Free Lunch* (Wolpert and Macready, 1997), em que é afirmado que todas as abordagens têm desempenho estatisticamente equivalente se forem considerados na análise *todos* os problemas possíveis. Em outras palavras, diferentes abordagens, com características e motivações distintas, tendem a ter melhor desempenho num nicho específico de problemas, mas, na média, considerando uma grande gama de problemas e com diferentes características, todas tendem a apresentar desempenho equivalente. Em problemas de otimização em que exista continuidade, diferenciabilidade e/ou estacionariedade, provavelmente métodos tradicionais, sustentados por algoritmos exatos e determinísticos, terão melhor desempenho que meta-heurísticas, por exemplo. Por outro lado, em problemas sem essas restrições, provavelmente meta-heurísticas populacionais terão melhor desempenho ou até nem sofrerão a concorrência de métodos tradicionais. No caso de GP, assim como em outras meta-heurísticas populacionais, as principais desvantagens são custo computacional e a falta de garantia de convergência para a solução ótima. Por outro lado, estas abordagens exigem menor conhecimento sobre o problema, suportam diferentes tipos de representação e, consequentemente, são flexíveis para atacar uma grande

gama de problemas (Fogel, 1997). A recente emergência de diversas arquiteturas computacionais, com múltiplos núcleos de processamento, favorece abordagens de GP, assim como outras abordagens populacionais, uma vez que são naturalmente paralelizáveis.

3.1 Representação dos programas

Programa de computador é uma sequência de instruções que é executada por um computador com intuito de realizar uma determinada tarefa. A arquitetura para computação mais utilizada é a de von Neumann (1945) que é subdividida em unidade de controle, unidade aritmética, memória de dados e programa, e dispositivos de entrada e saída. Basicamente, esta arquitetura permite que uma máquina execute instruções que operam informações armazenadas em memória e interaja com o meio externo através de dispositivos de entrada e saída. Analisando as diferentes arquiteturas de computador existentes, praticamente todas se baseiam nesse conceito. Até mesmo ambientes de execução virtuais, ou máquinas virtuais, em que o processo de execução de programas é simulado, seguem esta arquitetura de alguma forma, como será mostrado neste capítulo, uma vez que a maioria dos ambientes de programação genética opera utilizando instruções que são interpretadas e executadas num ambiente virtual.

O modo pelo qual programas são representados em GP afeta a ordem de execução de instruções, uso e localidade da memória e a aplicação de operadores genéticos (Banzhaf, 1998). As três formas mais utilizadas de representação de programas em GP são árvore, vetor (denominada representação linear) e grafo. Independentemente da representação, programas são compostos por operadores, constantes, variáveis, condições e laços. Este último nem sempre são suportados por abordagens de programação genética, uma vez que implica operadores genéticos mais complexos e métodos para garantir que não entrarão em laço infinito de execução.

O conjunto de instruções e variáveis que poderão ser utilizados pelos programas precisa ser definido de forma que satisfaça a condição de suficiência (Koza, 1992), garantindo a existência de uma determinada combinação de instruções e parâmetros no espaço de soluções que resolve o problema. É importante destacar que a definição do conjunto de instruções apresenta uma questão de compromisso entre expressividade da linguagem e tamanho do espaço de busca. Portanto, utilizar uma linguagem com alto nível de generalização, que atende a condição de suficiência para praticamente todos os problemas (linguagem Turing completa), pode implicar em evoluir programas num espaço de busca de soluções inviável para muitos dos problemas, dadas as limitações computacionais das arquiteturas modernas. Tanto que é comum em GP caminhar na direção contrária, ou seja, desenvolver conjuntos de instruções específicas para o tipo de problemas que será abordado, reduzindo drasticamente o espaço de busca e aumentando a interpretabilidade do problema. Estas questões relativas à concepção

de ambientes para indução de programas são abordadas no Capítulo 4, em que aplicações em diversas áreas são analisadas.

3.1.1 Representação em Árvore

Nesta representação, operadores e operandos estão dispostos em nós de uma árvore. A execução de um dado programa pode ser realizada de duas formas: percorrendo os nós em pós-ordem ou em pré-ordem. No primeiro caso, o programa é executado percorrendo os nós das extremidades até o nó raiz, ou seja, visitando primeiro os operandos e, em seguida, os operadores. No segundo caso, é realizado o processo inverso, parte-se da raiz e, para cada ramo, os nós são visitados até chegar aos nós folha. Esta última abordagem tem a vantagem de economizar recursos computacionais na existência de nós condicionais, uma vez que só serão executados os nós da subárvore associada à condição satisfeita.

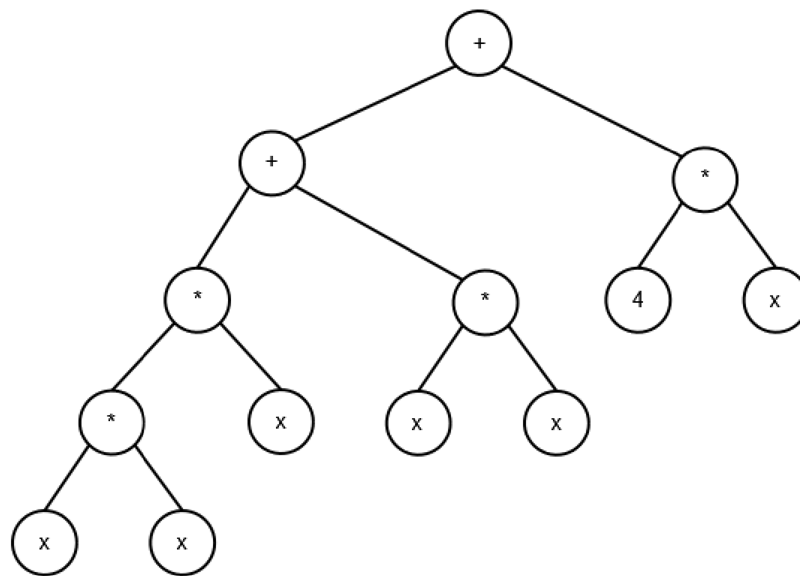

Fig. 3.1: Exemplo de programa representado em árvore que modela a função $f(x) = x^3 + x^2 + 4x$.

A Figura 3.1 mostra um exemplo de programa composto por 7 nós terminais, sendo que 6 estão vinculados à variável de entrada x e uma constante 4, que modela a função $f(x) = x^3 + x^2 + 4x$. Como pode ser observado, na representação em árvore, a memória do programa está na própria estrutura. Os nós podem acessar informações em nós terminais ou o retorno de um nó não-terminal. Sendo assim, o resultado de operações realizadas por uma subárvore só está acessível ao nó pai do nó raiz desta subárvore. Neste aspecto, existe uma desvantagem em relação a outras representações pois manipulações de memória comuns, contendo um subprocedimento útil a múltiplas partes do programa, precisam ser replicadas, ao invés de se armazenar o resultado da manipulação num endereço

de memória global, acessível a qualquer instrução do programa. Uma estratégia para contornar esta limitação, proposta por Koza (1994), é a definição automática de funções de programa. Semelhante a funções em linguagens de programação tradicional, este recurso permite que uma sequência de instruções úteis a múltiplas partes de um programa seja representada apenas uma vez na memória. Por um método de chamada de função, o fluxo de execução muda para o corpo da função, que realiza um determinado procedimento, e em seguida volta ao ponto de execução anterior. No caso de GP, uma função pode ser representada por uma subárvore do programa, tendo no nó raiz o rótulo da função. Toda vez que um nó do programa que referencia este rótulo é executado, a função é executada.

3.1.2 Representação Linear

Esta representação se assemelha a linguagens de programação imperativas como, por exemplo, a linguagem C, em que programas são representados por uma lista de instruções que são executadas em ordem. Para alterar o fluxo de execução, instruções condicionais mudam a instrução atual, podendo pular para qualquer trecho do programa. Dentre as representações de programas genéticos, está é a mais próxima da representação de programas executados pela maioria dos processadores. Inclusive, Nordin (1994) desenvolveu o sistema CGPS (do inglês, *Compiling Genetic Programming System*) que é capaz de gerar programas que podem ser executados diretamente em um processador, sem a necessidade de passar por um interpretador. Porém, na maioria dos casos, incluindo soluções representadas de forma linear, programas são interpretados e executados numa máquina virtual, uma vez que a representação utilizada e o conjunto de instruções utilizados pelos programas não são suportados pelos processadores tradicionais. De fato, é possível desenvolver ferramentas para converter programas nestes formatos para o formato utilizado pelos processadores. Porém, dada a evolução e a popularização das linguagens de alto nível, esta alterativa não tem despertado muito interesse dos pesquisadores.

Diferentemente da representação em árvore, na representação linear, instruções manipulam a memória diretamente, por meio de variáveis. Portanto, informações armazenadas em memória são de acesso global, podendo ser acessadas e modificadas por qualquer trecho do programa (Brameier and Banzhaf, 2007). Dentre as vantagens deste método, está a possibilidade de aproveitar resultados, obtidos por um determinado segmento de código do programa, em outras partes do programa.

A Figura 3.2 apresenta um programa em representação linear com o mesmo objetivo do programa apresentado na Figura 3.1, modelar a função $f(x) = x^3 + x^2 + 4x$. Neste caso, o resultado do programa é armazenado na variável de saída y, previamente definida.

```
a = x * x
b = a * x
c = 4 * x
y = a + b
y = y + c
```

Fig. 3.2: Exemplo de programa em representação linear que modela a função $f(x) = x^3 + x^2 + 4x$.

3.1.3 Representação em Grafo

Na representação em grafo, instruções são representadas por nós e arestas direcionadas, as quais determinam o fluxo do programa. Cada nó pode ter múltiplas arestas de entrada e múltiplas arestas de saída. Desta forma, é possível obter representações compactas de programas, uma vez que um mesmo fluxo de programa pode ser iniciado por diferentes nós. Cada nó tem uma ação e pode ter um condicional, que determina se um nó conectado por uma aresta será visitado.

O acesso à memória pode variar dentre diferentes implementações. No caso de PADO (do inglês, *Parallel Algorithm Discovery and Orchestration*) (Teller, 1996), por exemplo, que utiliza representação em grafo, uma pilha é utilizada como memória local e uma memória indexada é utilizada para armazenar variáveis globais. Nesta arquitetura, todas as instruções desempilham seus atributos e empilham o resultado, com exceção das instruções **read** e **write** que permitem acesso direto à memória indexada. A Figura 3.3 mostra um programa com o mesmo propósito dos programas apresentados nas seções sobre representação em árvore e linear, porém na representação em grafo utilizada pelo PADO.

No programa anterior, a execução começa pela instrução mostrada em cinza, e termina no nó "+" que não apresenta arestas de saída. Todos os nós têm uma condição para escolher dentre uma aresta ou outra, podendo utilizar informações na memória, na pilha ou sobre o nó anterior. No caso do exemplo anterior, as condições de cada nó apenas avaliam o tipo do nó anterior, dado o fluxo de execução do programa. Analisando este exemplo, fica claro que programas representados em grafo, naturalmente suportam laços e para evitar que entrem em laço infinito, o processo de execução pode ter um tempo ou um número máximo de instruções que, quando atingidos, a execução do programa é interrompida. Outras formas de representação de programas em grafo foram propostas: em Shirakawa et al. (2007), programas são capazes de manipular múltiplos tipos de dados, um mapeamento genótipo-fenótipo é utilizado para simplificar as operações genéticas que são realizadas num genótipo representado por uma sequência numérica; em Poli (1997), nós do grafo invocam a execução de múltiplos nós, possibilitando paralelismo pela execução de múltiplos nós concorrentemente; em Mabu (2007), nós

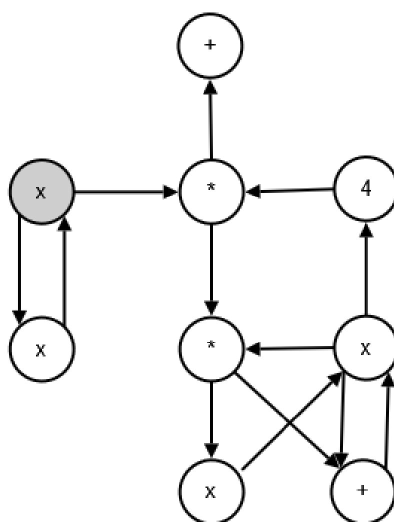

Fig. 3.3: Exemplo de programa representado em grafo que modela a função $f(x) = x^3 + x^2 + 4x$.

de julgamento e nós de ação são combinados para modelar o comportamento de agentes inteligentes.

3.2 Evolução de Programas

A seção anterior mostrou as diferentes formas comumente empregadas na representação de programas genéticos. Esta seção descreve as etapas do processo evolucionário que a partir de uma população inicial de programas, evolui soluções por meio de um processo iterativo, o qual seleciona com maior probabilidade os programas que melhor atendem as necessidades de um determinado problema, dada uma métrica. Independentemente da representação de programa adotada, o processo evolucionário é composto pelas mesmas etapas. A representação apenas implica na forma em que o programa é interpretado e manipulado, conforme será descrito. É evidente que a representação adotada determina o espaço de busca e as relações de vizinhança entre as soluções candidatas.

O algoritmo básico para evolução de programas é apresentado na Figura 3.4. O primeiro procedimento cria uma população de indivíduos. Na etapa de avaliação, cada indivíduo é executado, considerando apenas as instruções relevantes, e um valor de qualidade do indivíduo é determinado. Caso exista algum indivíduo na população atual que satisfaça o critério de parada, o processo de evolução é finalizado. Caso contrário, indivíduos de melhor qualidade são selecionados com maior probabilidade para serem submetidos aos operadores genéticos, gerando novas soluções. Estas novas soluções irão compor uma nova população de indivíduos, a qual será submetida aos mesmos procedimentos, com exceção do primeiro que cria a população inicial. A execução iterativa destes procedimentos permite a emergência de boas soluções, uma vez que, a cada geração, as melhores soluções têm maior probabilidade de serem selecionadas e alteradas. Assim como em outras meta-heurísticas populacionais,

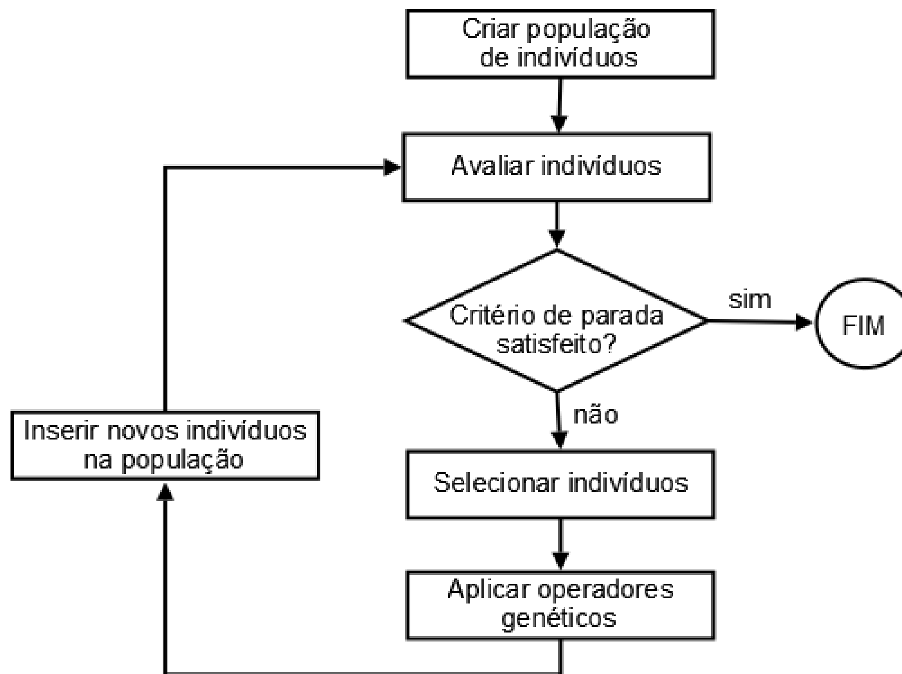

Fig. 3.4: Fluxograma do algoritmo de programação genética.

diversas partes do espaço de busca são exploradas ao mesmo tempo por diferentes indivíduos, que são guiados pela função de avaliação a regiões mais promissoras. Melhorias serão agregadas às soluções ao longo das gerações até que um critério de parada seja satisfeito.

Cada procedimento deste algoritmo será descrito com maior detalhe nas próximas seções.

3.2.1 Inicialização

O objetivo da inicialização é montar uma população inicial de indivíduos distintos capazes de explorar diversas regiões do espaço de soluções. Quando não há conhecimento *a priori* que sugira regiões mais promissoras do espaço de soluções, os indivíduos podem ser gerados aleatoriamente, portanto, tendendo a uma distribuição uniforme neste espaço. Por outro lado, quando conhecimento *a priori* sobre o espaço de soluções estão disponíveis, a geração de indivíduos em determinadas regiões pode receber uma maior probabilidade, aumentando o poder de exploração naquela região e, possivelmente, acelerando a convergência do processo evolucionário. Independentemente da representação de programa utilizada, normalmente este processo tem três parâmetros: tamanho da população inicial, tamanho mínimo e máximo do indivíduo. O primeiro parâmetro afeta a demanda de recursos computacionais por geração e a convergência do algoritmo. Os dois últimos parâmetros determinam a complexidade dos programas iniciais, considerando que quanto maior o número de instruções, maior a complexidade de um programa.

3.2.2 Avaliação

Indivíduos mais adaptados ao problema em questão ou, em outras palavras, as melhores soluções candidatas até a geração atual, são selecionados com maior probabilidade nesta etapa, para gerar descendentes para a próxima geração. Para determinar a qualidade de uma solução, é definida uma função de adaptação que determina o quão apto está um determinado indivíduo em relação ao problema em questão. Este índice de adaptação é também denominado *fitness*. A função de adaptação direciona a busca por indivíduos em regiões mais promissoras do espaço de soluções candidatas. No caso da aplicação de aprendizado supervisionado no processo de indução, pela utilização de casos de adaptação contendo exemplos de entrada e saída, podem ser computados o erro quadrático médio (EQM), no caso de problemas de predição ou aproximação de funções, e o erro de classificação, para problemas de classificação, considerando um dado programa. Os casos de adaptação contêm um vetor de entrada $\vec{i}$ e um vetor de saída $\vec{o}$ para n exemplos. No caso do EQM, é calculado o somatório da diferença entre a saída do programa e a saída desejada, para os n exemplos, elevado ao quadrado. Este valor é divido pelo número de amostras para se obter uma média, conforme a Equação 3.1:

$$EQM_{prog} = \frac{1}{n}\sum_{k=1}^{n}(prog(i_k) - o_k)^2. \tag{3.1}$$

Já no caso de problemas de classificação, é calculado o número de amostras classificadas incorretamente, conforme a Equação 3.2:

$$EC_{prog} = \sum_{k=1}^{n}\{1|class(prog(i_k)) \neq o_k\}. \tag{3.2}$$

Usualmente, a qualidade das soluções é representada no intervalo $[0, 1]$, podendo ser calculada em função do erro ou qualquer outra métrica, sendo que 1 indica a solução ótima e valores positivos menores que 1 indicam o quão distante um dado candidato está da solução ótima, conforme a Equação 3.3:

$$Adaptacao_{prog} = \frac{1}{1 + erro}. \tag{3.3}$$

Entretanto, é importante ressaltar que existe uma grande diversidade de opções para a função de avaliação. Para possibilitar a identificação de boas soluções, esta função deve ser capaz de distinguir diferentes graus de qualidade de soluções. Tendo uma superfície de *fitness* com essa diferenciação, indivíduos irão amostrá-la pela aplicação de operadores genéticos, em busca de regiões promissoras, ou seja, regiões que contêm indivíduos com altos índices de adaptação, de acordo com a Equação 3.3. No capítulo 4, diferentes aplicações de GP são apresentadas, incluindo diferentes formas de avaliar a qualidade de um indivíduo.

3.2.3 Seleção

Tendo um grau de qualidade para cada indivíduo da população, pode-se empregar diferentes estratégias de seleção para determinar quais indivíduos serão utilizados para gerar novas soluções. Na estratégia elitista, por exemplo, apenas os n melhores indivíduos são selecionados. Apesar de parecer intuitiva, esta estratégia pode polarizar o processo de busca às melhores soluções das primeiras gerações e acelerar o processo de perda de diversidade genética na população. Sendo assim, outras estratégias mantém parte dos indivíduos que não estão entre as melhores soluções da geração atual, mas que, com a aplicação dos operadores genéticos, podem resultar em boas soluções nas gerações futuras.

Na seleção por roleta, um indivíduo i com *fitness* f_i tem a probabilidade p_i de ser selecionado, numa população com n indivíduos, proporcional ao seu grau de aptidão, como mostrado na equação 3.4. Desta forma, mesmo com a menor probabilidade dentre todos os indivíduos, o pior indivíduo pode ser selecionado.

$$p_i = \frac{f_i}{\sum_{k=1}^{n} f_k}. \tag{3.4}$$

Com o intuito de prover um maior controle sobre as probabilidades de seleção dos indivíduos, foi proposto o método de seleção por *ranking*. Neste método, após ordenar a população de indivíduos pelo grau de aptidão (criando-se um *ranking*), a probabilidade de um indivíduo ser selecionado é modelada em função de sua posição no *ranking*. Utilizando funções de distribuição de probabilidades linear ou exponencial, por exemplo, é possível controlar, manipulando os parâmetros destas funções, a diferença das probabilidades de seleção entre os melhores, piores e indivíduos medianos.

Outra estratégia bastante utilizada é a seleção por torneio. Nesta estratégia, k indivíduos são selecionados aleatoriamente para a composição do conjunto de indivíduos que participarão do torneio. Deste conjunto, indivíduos são selecionados em função de seu *fitness*, denominados vencedores, para substituir os piores indivíduos. Manipulando o valor de k é possível controlar a pressão seletiva. Quanto menor este valor, menor a pressão e vice-versa. Na situação em que $k = 1$, o processo seletivo é equivalente a seleção aleatória.

3.2.4 Reprodução

A reprodução de um indivíduo consiste na adição de um ou mais indivíduos na população, contendo o mesmo código genético. O número de novos indivíduos que serão adicionados para cada indivíduo selecionado é determinado pela taxa de reprodução. A pressão exercida pelo processo de seleção, ou seja, a porcentagem de indivíduos selecionados naquele processo, em conjunto com a taxa de reprodução, influencia diretamente a velocidade de convergência e a perda de diversidade

(Brameier and Banzhaf, 2007). Taxas altas de reprodução podem levar o processo evolutivo a uma convergência prematura pela rápida perda de diversidade. Por outro lado, a seleção de um número maior de indivíduos e a utilização de taxa menor de reprodução podem atrasar a convergência do processo evolucionário, uma vez que a cada geração, parte significante da população está explorando regiões menos promissoras, dado o conhecimento adquirido até a geração atual. Porém, é importante ressaltar que estes parâmetros estão diretamente associados a questões específicas do problema a ser abordado. Uma análise completa, ou de subespaços, da superfície de adaptação pode auxiliar a escolha de valores para esses parâmetros, visando atender características específicas do problema em questão. Os indivíduos resultantes da reprodução sofrerão variação a partir de operadores genéticos. Elas podem ser de dois tipos: recombinação e mutação. Esses tipos de variação serão devidamente apresentados nas próximas subseções.

Recombinação

Uma vez que uma subpopulação de indivíduos, com maior propensão de solucionar o problema em questão, foi selecionada e reproduzida, o próximo passo é promover variação nesta população. Este processo promove a exploração de novas soluções que serão avaliadas e selecionadas na próxima iteração do processo evolucionário.

O operador de recombinação realiza a troca de material genético entre dois indivíduos reproduzidos a partir do conjunto de indivíduos selecionados na geração anterior. Entretanto, é comum combinar a etapa de reprodução e recombinação numa única etapa, em que novos indivíduos já são gerados com a operação de recombinação realizada. Independentemente da representação, o princípio é o mesmo: selecionar uma parte do código genético de cada indivíduo progenitor e intercambiá-las para gerar indivíduos filhos. A maneira mais comum de utilização deste operador é a geração de dois novos indivíduos, denominados descendentes, a partir de dois indivíduos progenitores.

A Figura 3.5 mostra a aplicação do operador de recombinação a dois indivíduos representados em forma de árvore, para gerar dois novos indivíduos. Nos indivíduos **A** e **B**, foi selecionada aleatoriamente uma subárvore, conforme destacado, e, em seguida, dois novos indivíduos, **A’** e **B’**, são gerados trocando essas subárvores.

No caso da representação linear, são selecionadas uma ou mais sequências de instruções em cada indivíduo pai para realizar o intercâmbio. Uma vez que o objetivo do operador de recombinação é possibilitar a troca de segmentos de código genético responsáveis por atuar numa parte do problema em questão, não é interessante permitir a seleção de segmentos com apenas uma instrução ou segmentos com quase todas as instruções do indivíduo. Sendo assim, no processo de seleção de segmento de código, é recomendável definir os tamanhos mínimo e máximo dos segmentos. A Figura 3.6 ilustra o processo de recombinação envolvendo indivíduos em representação linear.

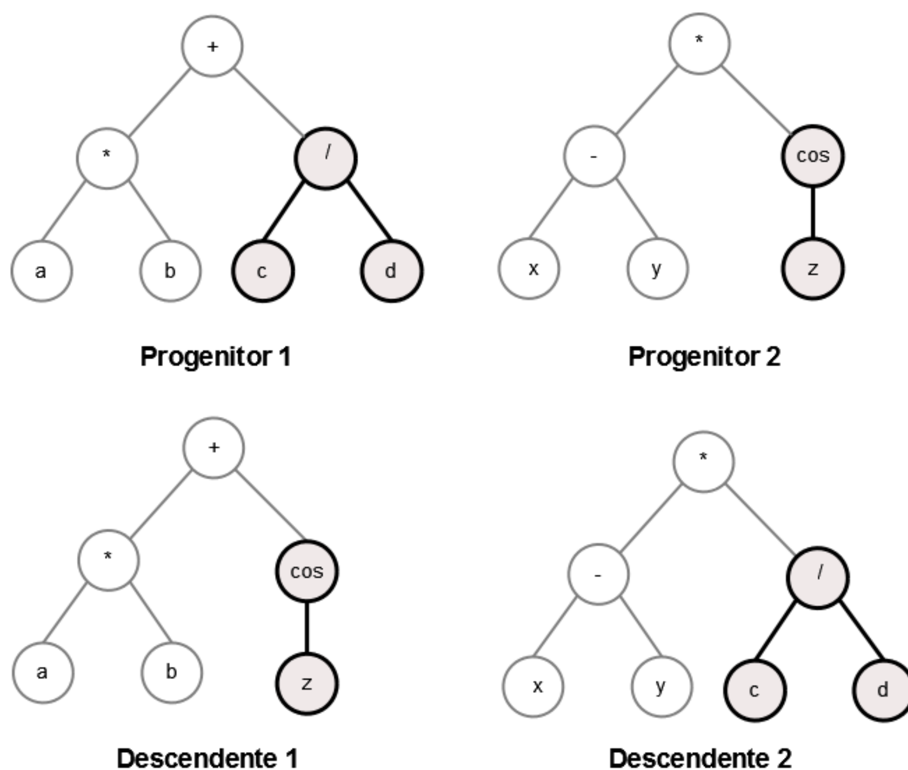

Fig. 3.5: Operador de recombinação para representação em árvore.

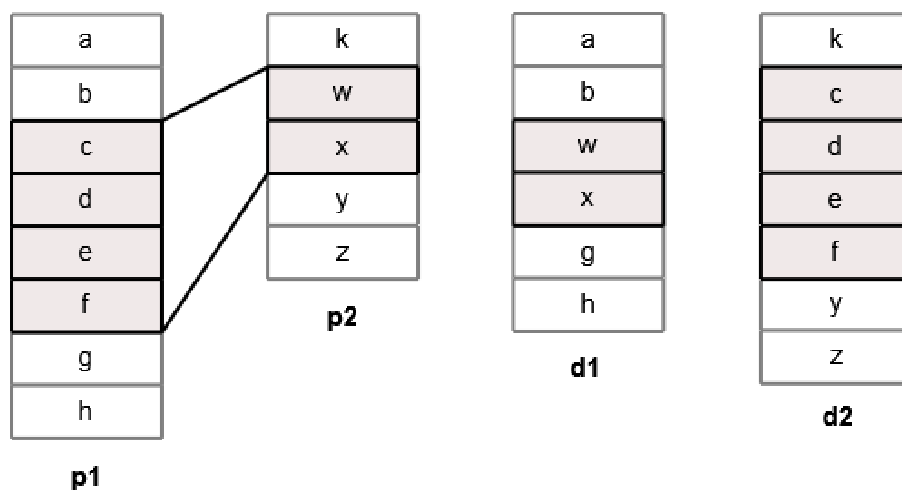

Fig. 3.6: Operador de recombinação para representação linear.

Recombinação de programas representados como grafos não é tão intuitiva como no caso das representações em árvore ou linear. Teller (1996) propôs o seguinte procedimento:

1. Selecione dois indivíduos progenitores.

2. Divida cada indivíduo em dois conjuntos de nós (fragmentos).

3. Marque as arestas como “interna” caso apontem para nós no mesmo fragmento.
4. Marque as arestas como “externa” caso apontem para nós no outro fragmento.
5. Troque fragmentos entre os dois programas.
6. Aponte todos as arestas marcadas como “externa” à nós do novo fragmento, selecionados aleatoriamente.

Mutação

O operador de mutação altera parte do código genético de um indivíduo. Este operador simula o mesmo mecanismo encontrado na natureza, em que indivíduos têm seus códigos genéticos alterados, por exemplo, por falha na replicação de parte de seu código genético.

Em GP, o operador de mutação pode realizar as seguintes operações:

- alterar tipo de instrução;
- Alterar parâmetros de uma instrução;
- Alterar valor de uma constante;
- Alterar a posição de uma instrução;
- Adicionar instruções geradas aleatoriamente;
- Remover instruções geradas aleatoriamente;

Estas operações podem ter diferentes probabilidades de serem escolhidas no processo de mutação. Inclusive, estas probabilidades podem ser controladas para mudar o comportamento da população ao longo do processo evolucionário. Afinal, um problema que pode ocorrer num processo evolucionário de GP é o crescimento excessivo de tamanho dos programas. Este crescimento pode ocorrer pelo acúmulo de instruções que não afetam a execução do programa (introns) ou pelo desenvolvimento de soluções de tamanho excessivo desnecessário (Langdon and Poli, 1998). O primeiro fenômeno ocorre principalmente pelo fato de que, quanto maior o número de introns, menor a probabilidade dos operadores genéticos destruírem blocos construtivos do programa - sequências de instruções que resolvem parte do problema. Sendo assim, num estado maduro do processo evolucionário, em que a probabilidade de um operador genético maximizar o *fitness* de um indivíduo é bem menor do que nas gerações iniciais, o mais provável é o acúmulo deste tipo de instrução. O outro fenômeno, desenvolvimento de soluções de tamanho excessivo desnecessário, também ocorre pela mesma pressão imposta pelo método de avaliação. Num determinado estágio do processo evolucionário, pode ocorrer que a probabilidade de encontrar uma solução com mesmo valor de *fitness*, porém com um número maior de instruções, seja maior do que encontrar uma solução com um maior valor de *fitness*. Quando

há incidência deste cenário, a tendência é o desenvolvimento de soluções de tamanho desnecessário, uma vez que soluções mais compactas conseguem obter o mesmo valor de *fitness*. É importante ressaltar que estes fenômenos ocorrem caso nenhum mecanismo de controle seja aplicado. Uma forma de minimizar este fenômeno é a punição do tamanho do programa no método de avaliação, induzindo a obtenção de soluções promissoras e com um maior grau de parcimônia.

3.2.5 Detecção de Introns

Em GP, intron é qualquer instrução ou sequência contígua de instruções que não influencia o resultado da execução de um programa, dada uma função de avaliação. Estas instruções surgem no processo evolutivo como um mecanismo de proteção, que reduz a probabilidade de destruição de blocos construtivos - sequências de instruções que resolvem parte do problema - pelos operadores genéticos (Brameier and Banzhaf, 2007).

No caso da programação genética linear, Bramier e Banzhaf (2001) propuseram a diferenciação de introns em duas categorias: estruturais e semânticos. O primeiro corresponde às instruções individuais que manipulam variáveis que não afetam o resultado do programa, dado o trecho atual de código. O segundo corresponde a uma instrução ou uma sequência contígua de instruções que manipula registradores que afetam o resultado do programa, porém de forma que os valores destes registradores continuam os mesmos ao final, por exemplo, realizando a operação **A** e, em seguida, a operação inversa **A'**.

Visto que estas instruções não afetam o resultado de execução de um programa, pode-se economizar recursos computacionais simplesmente não as executando. Porém, estas instruções devem ser mantidas nos indivíduos da população para continuarem desempenhando o seu papel de mecanismo de proteção às operações de destruição de blocos construtivos, de forma similar ao que aparentemente ocorre na natureza.

3.3 Programação Genética Fortemente Tipada

Nas representações mostradas nas seções anteriores, instruções operam manipulando informações armazenadas na própria estrutura do programa, ou em registradores que podem ser acessados por qualquer instrução do programa. Entretanto, não existe nenhum tipo de diferenciação de dados armazenados e manipulados pelos programas, diferentemente da maioria das linguagens de programação que contam com tipagem de dados. Nestas linguagens, instruções podem ser implementadas contendo parâmetros com tipos específicos, como valores inteiros ou de ponto flutuante, vetores, matrizes, etc.

Para incorporar estes conceitos na indução automática de programas, Montana (1995) introduziu o conceito de Programação Genética Fortemente Tipada (STGP, da sigla em inglês). Montana utilizou a representação de programas em árvore, mas o conceito é válido para qualquer representação. Utilizando este conceito, além do número de parâmetros, cada instrução tem também o tipo de cada parâmetro, podendo ser tanto de tipos primitivos como, por exemplo, números inteiros ou reais, ou até estruturas de dados como, por exemplo, vetores, matrizes ou estruturas mais complexas. Nos casos em que programas podem utilizar registradores de diversos tipos, é garantido em todos os operadores genéticos que as instruções terão todos os seus parâmetros com o tipo correto, removendo do espaço de busca programas parametrizados incorretamente e acelerando o processo evolucionário.

3.4 Considerações Finais

Neste capítulo, foi abordada a técnica mais utilizada atualmente para geração automática de programas por meio de computação evolucionária. Portanto, esta técnica permite a emergência de uma solução não prevista previamente, diferentemente de ferramentas em que programas de computador são gerados a partir de uma solução previamente modelada por um ser humano num outro determinado formato, como é o caso de ferramentas de programação automática (Balzer, 1985).

Programação genética é uma abordagem de computação evolucionária para a solução de problemas de propósito geral. Uma vez que são suportados diferentes representações de problemas e conjunto de instruções, GP pode ser aplicada a uma grande gama de problemas. Em alguns casos, o produto final é o próprio programa, por exemplo, um controlador robótico. Em outros casos, o programa é uma ferramenta que, quando executada, sintetiza o produto final, por exemplo, um circuito elétrico analógico. Esta versatilidade permite o emprego de diferentes técnicas de aprendizagem de máquina para representar, selecionar e avaliar soluções candidatas, o que ficará evidente no Capítulo 4, em que aplicações de GP em diversas áreas serão analisadas.

Capítulo 4

Aplicações de Programação Genética

Por incorporar naturalmente a evolução de estruturas e parâmetros de modelos de representação de soluções candidatas, programação genética (GP, do inglês *Genetic Programming*) é uma das abordagens mais versáteis de aprendizado de máquina. Do ponto de vista teórico, utilizando uma linguagem Turing completa para a representação de programas genéticos, está presente no espaço de soluções candidatas qualquer outra abordagem de aprendizado de máquina que pode ser representada como um programa de computador. Este fato apenas demonstra o grau de flexibilidade que pode-se obter com GP. Porém, do ponto de vista prático, a existência de uma solução no espaço de soluções não implica que é viável encontrá-la. No intuito de trabalhar num espaço de soluções em que é viável a indução de programas genéticos, é comum a utilização de linguagens específicas e restritas ao domínio dos problemas para os quais foram propostas. Sendo assim, não é comum a utilização de um *framework* genérico para problemas de GP. Em muitos casos, ambientes completos de indução de programas, contendo um conjunto de instruções e estruturas de dados específicas, são desenvolvidos para atacar um único problema.

Versatilidade na representação de soluções candidatas permite o emprego de GP para os mais diversos problemas de aprendizado de máquina, de robótica autônoma a mineração de dados. Do ponto de vista de estratégias de aprendizado de máquina, GP também se mostra aberta à utilização de diversas abordagens. No caso de aprendizado de máquina a partir de bases de dados, programas genéticos podem ser induzidos por meio de processos de aprendizado supervisionado, não-supervisionado e semi-supervisionado. No primeiro caso, as amostras utilizadas no processo de treinamento contêm um conjunto de atributos e um rótulo que descreve o resultado desejado em função deste conjunto. No caso de um problema de classificação, o conjunto de atributos contém propriedades que permitem discriminar amostras de diferentes classes e o rótulo representa a classe da amostra. No caso de um problema de regressão de função, ao invés de uma classe, o rótulo armazena o valor que a função retorna com os parâmetros armazenados no conjunto de atributos. Por sua vez, o aprendizado não-

supervisionado utiliza amostras sem rotulação. Neste caso, a abordagem busca encontrar estruturas desconhecidas presentes na base de dados, como, por exemplo, agrupamentos de amostras. O fato de haver este tipo de estrutura presente nos dados indica que há uma correlação entre amostras de uma mesma classe. Finalmente, no caso do aprendizado semi-supervisionado, são utilizadas tanto amostras com rótulo como amostras sem rótulo. As amostras com rótulo, normalmente presentes em menor quantidade, são utilizadas para aumentar a acurácia do modelo ou inferir as classes de um conjunto bem maior de amostras sem rotulação.

Visto que a definição de uma abordagem que empregue GP implica em decisões relativas ao conjunto de instruções, estruturas de dados, representação de programas, estratégias de aprendizado, métodos de avaliação, dentre outras, este capítulo analisa diversas aplicações de GP. As aplicações apresentadas neste capítulo são utilizadas como referência para a proposição de um método para a geração automática de programas que visam a modelagem de processos de manipulação de informação, apresentado no capítulo seguinte. De acordo com a similaridade entre algumas aplicações e a correlação com a proposta deste trabalho, os exemplos de aplicação são analisados em diferentes profundidades.

4.1 Síntese de Circuitos Elétricos

Existem problemas em que soluções não podem ser representadas diretamente como programas de computador. Porém, nada impede a indução de um programa que seja capaz de sintetizar uma solução para o problema. Este tipo de abordagem foi utilizada, por exemplo, para a síntese de circuitos elétricos, em que o fenótipo de uma solução candidata é um programa capaz de sintetizar um circuito, utilizando instruções que adicionam componentes ao circuito, ao invés de ser o próprio circuito. A vantagem desta abordagem é a abrangência de problemas que podem ser tratados por programas sintetizadores de solução.

Utilizando o circuito embrionário mostrado na Figura 4.1 em que, com exceção de **Z0** e **Z1**, todos os elementos são fixos e não podem ser modificados durante o processo de síntese do circuito, Koza et al. (1996) definiu uma abordagem para síntese de circuitos para mapear uma relação de entrada e saída. Cada programa candidato tem um nó raiz **List** com dois nós filho, **C** e **FLIP**, que são o nó inicial de dois subprogramas que sintetizam estruturas que substituem os elementos **Z0** e **Z1**, respectivamente. As placas são representadas num formato digital, podendo ser testadas e validadas no simulador *SPICE (Simulation Program with Integrated Circuit Emphasis)*. Na etapa de avaliação, todos os indivíduos são executados, cada um gerando subestruturas de circuito que substituem partes da placa embrionária. Numa etapa seguinte, a representação da placa é simplificada. Quando possível, fios que ligam dois elementos são removidos e estes são fundidos. Outra simplificação é

a remoção de subestruturas isoladas. Finalmente a placa pode ser avaliada no simulador utilizando um conjunto de casos de avaliação, informando a voltagem de entrada e a voltagem de saída. Os resultados mostram que a abordagem é competitiva quando comparada com o método manual de projeto de circuitos. Além do mais, dentre as diversas placas geradas automaticamente, em alguns casos emergiram soluções de topologia conhecida no universo de projeto de circuitos.

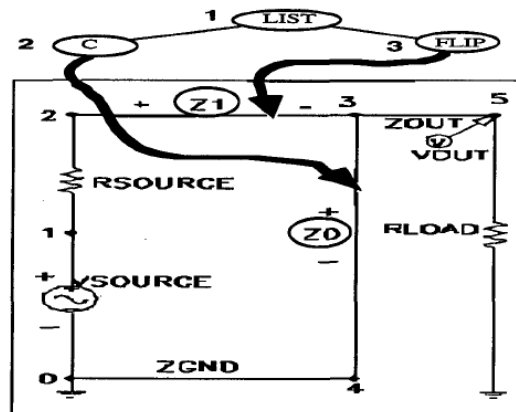

Fig. 4.1: Circuito embrionário utilizado para a síntese de circuitos elétricos (Koza, Bennett III, Andre and Keane, 1996).

Em trabalhos seguintes, o grupo de pesquisa liderado por Koza continuou a explorar as possibilidades de síntese de circuitos elétricos utilizando GP. Em Koza et al. (1996), um método automático para definição de funções (ADF, do inglês *Automatically Defined Functions*) é aplicado para permitir o reuso de partes do programa sintetizador, permitindo que subprogramas para síntese de sub-circuitos sejam utilizados múltiplas vezes por uma mesma solução. Em Koza et al. (1997), utilizando a abordagem que acrescenta ADFs, foram sintetizados circuitos capazes de resolver 8 problemas distintos, mostrando a abrangência da abordagem.

4.2 Robótica

A diminuição de custos de produção e miniaturização de componentes vem aumentando os investimentos na área de robótica. Apesar de não ser comum, atualmente é possível ter um robô autônomo realizando tarefas domésticas simples. A *IRobot*, empresa mais popular no seguimento de robôs domésticos, oferece soluções para aspiração e limpeza de chãos, piscinas e calhas. Mesmo que a aplicação doméstica ainda seja singela, se comparada com aplicações de robótica na indústria de manufatura, o início da comercialização de robôs para esse propósito sinaliza uma nova tendência que deve se popularizar nas próximas décadas. Tudo indica que cada vez será mais comum a presença de robôs em lares e empresas que não atuam no seguimento de manufatura. Tanto que empresas como

Microsoft e *Google* têm seus próprios departamentos de robótica. Toda essa expectativa aumentou ainda mais a demanda por projetos de pesquisa na área de robótica, incluindo a área de controle, junto a qual muitas abordagens de inteligência computacional têm sido aplicadas, inclusive GP.

4.2.1 Navegação Autônoma

As primeiras aplicações comerciais de robótica que obtiveram sucesso foram na área de automação industrial. Braços robóticos articulados têm sido aplicados com sucesso em diversas áreas, sendo a de maior destaque a manufatura de produtos como placas eletrônicas e veículos. A mais incomum talvez seja o manuseio de carga em ônibus espacial (Gillett et al., 2004). Porém, robôs deste tipo atuam totalmente dependentes de uma interação humana, seja por programação prévia ou teleoperação.

A necessidade de robôs com algum grau de autonomia fez com que emergisse um novo campo de estudo, o desenvolvimento de robôs autônomos. Autonomia, neste caso, não significa apenas a capacidade de realizar tarefas sem acompanhamento de um ser humano. Muitos dos robôs utilizados na área de manufatura atendem esse requisito. Autonomia é a capacidade de tomar decisões em diferentes cenários, com algum grau de independência do ser humano. Visto que estar fixado fisicamente num local restringe a capacidade de atuação do robô no ambiente, é comum que este tipo de robô tenha capacidade de se locomover autonomamente. Portanto, pode-se resumir que robôs com este tipo de autonomia precisam ter a capacidade perceber o ambiente por meio de sensores; calcular a forma de atuar no ambiente para obter um determinado resultado; mapear, se localizar e planejar a navegação pelo ambiente (Siegwart and Nourbakhsh, 2004).

Programação genética tem sido utilizada para abordar problemas na área de robótica autônoma. Na área de navegação, um problema comum é navegar pelo ambiente desviando de obstáculos. Para este tipo de problema, o conjunto de instruções de programa deve ser capaz tanto de analisar os valores dos sensores do robô como permitir a locomoção do mesmo. Neste tipo de aplicação, uma técnica comumente aplicada no processo de evolução é um sistema de recompensa baseado nos valores dos sensores e na locomoção do robô. Toda vez que o robô se locomove, ele recebe uma recompensa positiva. Por outro lado, toda vez que o robô colide ou chega muito próximo de um obstáculo, ele recebe uma recompensa negativa. A recompensa é utilizada na função de *fitness* para determinar a qualidade das soluções. O processo de aprendizagem pode ser realizado tanto de forma *online* como *offline*. No primeiro caso, a cada iteração, o robô atua no ambiente e recebe uma avaliação por essa atuação. Portanto, o processo evolutivo faz parte do controlador. No segundo caso, o robô atua no ambiente e é avaliado considerando um determinado intervalo de tempo de operação. Portanto, o algoritmo evolutivo é utilizado para projetar o controlador e não fará parte da solução final. O aprendizado *online* pode ser verificado nos trabalhos de Nordin e Banzhaf (1997a; 1997b), em que

programas genéticos foram gerados para controlar um robô *Khepera* capaz de navegar pelo ambiente desviando de obstáculos. Por outro lado, em Dain (1998) foi utilizado aprendizado *offline* para a geração de controladores de um robô capaz de navegar pelo ambiente seguindo as paredes.

4.2.2 Robocup

Robocup é uma competição internacional de robótica que promove pesquisa e educação na área de inteligência artificial, por meio do desenvolvimento de robôs autônomos capazes de jogar futebol. A meta oficial deste projeto é ter, na metade deste século, um time de robôs capazes de vencer o último vencedor da Copa do Mundo. Não há restrição com relação à abordagem utilizada para controlar o robô. Luke (1998) evoluiu programas genéticos para controlar jogadores de um time de robôs, utilizando um conjunto com 29 instruções, algumas genéricas como rotacionar 180 graus e outras mais específicas para jogar futebol, por exemplo, instruções para chutar ao gol, tentar um drible, e tocar a bola para o companheiro posicionado mais ofensivamente. O genótipo de cada solução candidata contém subprogramas para cada jogador do time, ou seja, cada jogador tem um programa específico. Cada jogador tem três estados: **chutar**, **andar**, **girar**. O robô só entra nos dois primeiros estados caso consiga ver a bola, sendo que entra no primeiro caso esteja perto da bola o suficiente para chutar, e no segundo estado, caso contrário. Para cada um desses dois estados, existe um programa distinto que determina a ação do robô. Quando o robô não consegue ver a bola, ele entra no terceiro estado, em que automaticamente gira até encontrá-la. Para evoluir esses jogadores, foi criado um torneio virtual, utilizando o simulador mostrado na Figura 4.2, em que soluções candidatas jogam umas contra as outras e o método de avaliação é o número de gols na partida. Este trabalho é considerado um marco na área de programação genética, tanto que foi premiado pelos organizadores do *Robocup* (Hedberg, 1997).

4.3 Descoberta de Conhecimento

Descoberta de Conhecimento é um campo de estudo que engloba um grupo de técnicas utilizadas para extrair conhecimento de conjuntos de dados. Basicamente, busca-se extrair um sentido ou explicação para determinadas relações entre elementos que fazem parte de uma base de dados, denominado conhecimento. Problemas deste tipo podem ser encontrados em praticamente todas as áreas da ciência, desde astronomia a ciências sociais. Na área médica, uma aplicação comum é tentar encontrar uma relação entre os sintomas dos pacientes e os diagnósticos. Tendo esta relação extraída de uma base de dados de histórico de pacientes, é possível auxiliar médicos na elaboração de novos diagnósticos. No setor financeiro, uma aplicação comum é a modelagem da relação entre atributos de clientes

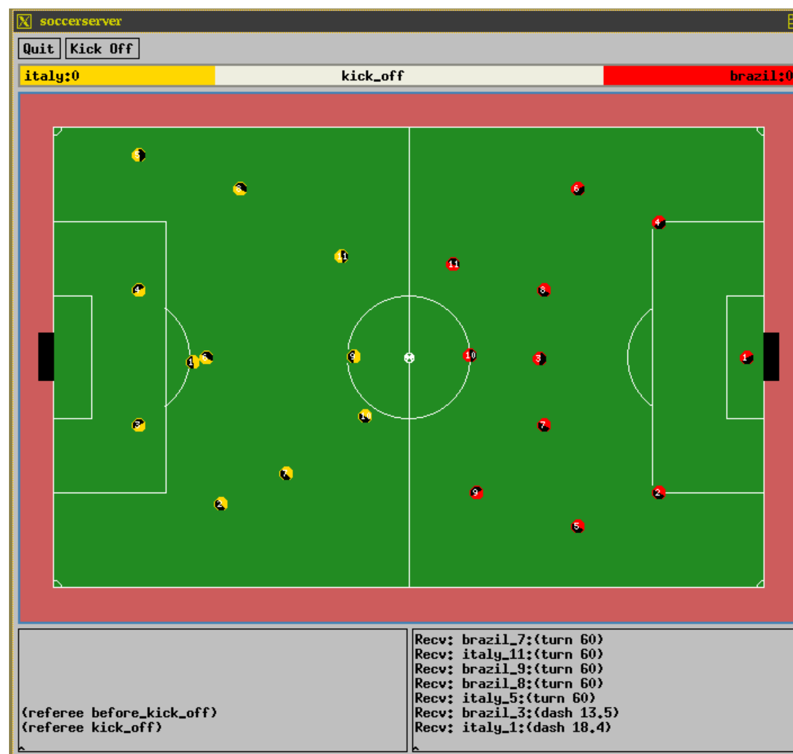

Fig. 4.2: Simulador oficial da competição *Robocup*

.

e inadimplência, com o intuito de diminuir os riscos em concessões de crédito. Portanto, qualquer área que tenha processos que produzam dados com propriedades que não são óbvias e intuitivas de serem extraídas pode utilizar técnicas de descoberta de conhecimento para tal propósito.

Entretanto, dos dados brutos à extração de conhecimento existem diversas etapas que precisam ser realizadas (Han and Kamber, 2006), sendo que uma proposta bem aceita é aquela apresentada na Figura 4.3. Partindo de um conjunto de dados armazenados num banco de dados ou uma base de dados linear, são realizadas a limpeza dos dados e integração, podendo dar origem a um *Data Warehouse* (DW). Entretanto, o uso de um DW não é obrigatório. Em seguida, na etapa de seleção e transformação, os dados são pré-processados para a etapa de mineração de dados. Na etapa de mineração de dados, padrões presentes no conjunto de dados são identificados e modelados. Na etapa seguinte, estes padrões são apresentados e avaliados. A análise destes padrões pode descrever fenômenos até então desconhecidos, em relação ao conjunto de dados analisado, resultando, portanto, num conhecimento novo. A seguir, estas etapas são descritas em maiores detalhes.

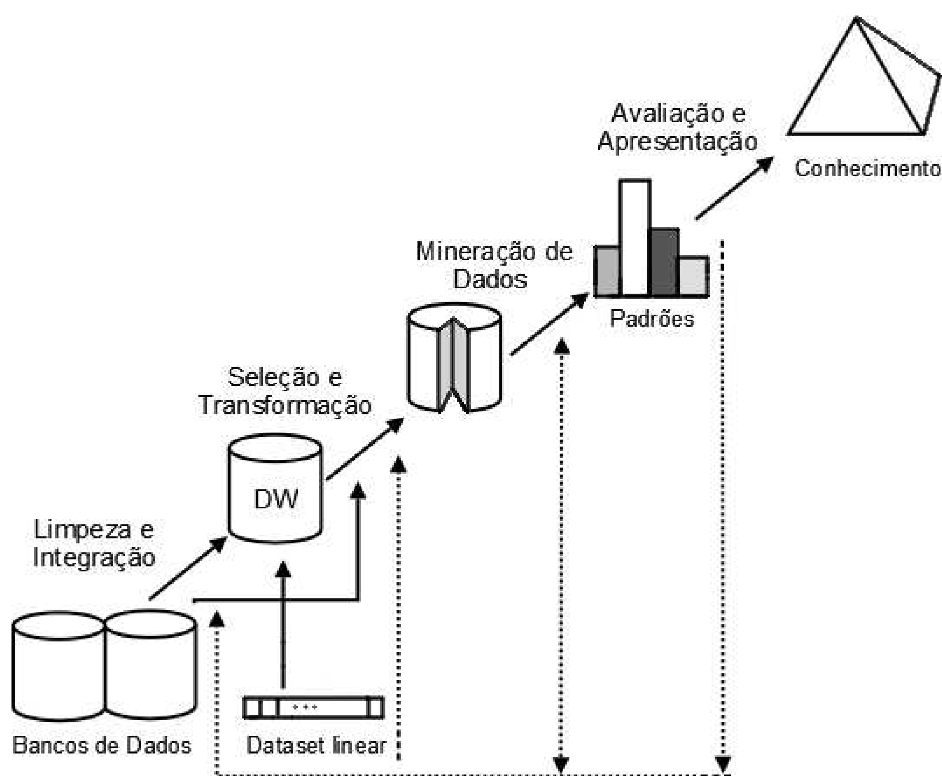

Fig. 4.3: Etapas do processo de descoberta de conhecimento, adaptado de Han & Kamber (2006).

Limpeza dos Dados

Nesta primeira etapa, o objetivo é realizar a limpeza dos dados para tratar dados faltantes, remover ruídos e inconsistências. Registros com dados faltantes podem ser descartados ou tratados através da imputação. A quantidade de ruído presente nos dados pode ser reduzida por técnicas de suavização, por exemplo, quantização, agrupamento e aproximação de funções. Esta etapa pode ser determinante em relação ao sucesso da aplicação, uma vez que o conhecimento irá emergir a partir dos dados, baseado em padrões que podem estar camuflados em meio a dados faltantes, ruídos ou inconsistências.

Integração dos Dados

A descoberta de conhecimento relacionada a um processo ou atividade pode envolver múltiplas fontes de informação. Nesta etapa, o objetivo é criar uma base de dados unificada e coerente. Dentre os desafios de integração de bases de dados, está o casamento correto de registros de fontes distintas, redução de redundância e resolução de conflitos entre registros.

Seleção dos Dados

Nesta etapa, é determinado se todo o conteúdo da base de dados será utilizado para descoberta de conhecimento. Em algumas situações, considerando bases de dados com grande volume de informação (e.g., centenas de milhares de registros), é necessário reduzir o número de registros para viabilizar algumas tarefas de mineração de dados que, dependendo da abordagem empregada, apresentam restrições em relação ao tamanho da base de dados. Para este propósito, podem ser empregadas técnicas de amostragem para produzir uma base de dados de menor volume e com as mesmas propriedades estatísticas. Entretanto, em outros casos, o problema não é o número de registros, mas sim o número de atributos. Atributos desnecessários, que não estão correlacionados com o tipo de conhecimento que se deseja obter, devem ser identificados e removidos para permitir uma representação de conhecimento de forma mais objetiva. O volume de dados na base de dados também pode ser reduzido pelo emprego de técnicas de quantização de amostras (Ng and Ravishankar, 1995) e generalização em que um conjunto de atributos pode ser transformado num atributo único. Em outros casos, todos os atributos são relevantes, mas a alta dimensão dos dados (número de atributos) prejudica a etapa de mineração de dados. Neste último caso, técnicas de redução de dimensão podem ser aplicadas, como por exemplo, análise de componentes principais (Pearson, 1901).

Transformação dos Dados

A última adequação dos dados para o processo de mineração é realizada nesta etapa. Por meio da operação de **agregação**, registros podem ser combinados, por exemplo, para que informações de diárias ou mensais de um determinado processo sejam transformadas em informações anuais. Na operação de **generalização**, dados brutos são substituídos por conceitos de maior nível obtidos pela utilização de hierarquia de conceitos (Han and Fu, 1994), por exemplo, datas podem ser categorizadas em *feriado*, *dia útil* ou *final de semana*; coordenadas geográficas podem ser trabalhadas em diferentes níveis: *rua*, *bairro*, *cidade*, *estado*, *país* e *continente*. A operação de **normalização** é utilizada quando se deseja dimensionar os valores dos atributos dentro de um intervalo específico, por exemplo, entre -1 e $+1$. Finalmente, a operação de **construção de atributos** é utilizada para criar novos atributos a partir dos atributos existentes. Os atributos *receita*, *custos*, *impostos* podem ser combinados para gerar o novo atributo *lucro*.

Mineração de Dados

Todas as etapas anteriores foram dedicadas à preparação dos dados para a descoberta de conhecimento baseada em padrões presentes nos dados. Finalmente, na etapa de mineração, estes padrões são capturados por algoritmos de estatística e inteligência computacional, tendo como produto deste

processo um modelo que armazena as informações que definem a relação entre elementos da base de dados, denominado conhecimento. Dada a sua importância, esta etapa é tratada à parte na seção 4.4.

Avaliação dos Padrões

Técnicas de mineração de dados são capazes de encontrar diversos padrões presentes na base de dados. Entretanto, antes de apresentar o conhecimento extraído da base de dados, é importante analisar os padrões obtidos, uma vez que nem todos os padrões são relevantes. Algumas características que tornam um determinado padrão interessante para a descoberta de conhecimento são (Han and Kamber, 2006):

- Grau de interpretabilidade para um ser humano. Pouco adianta extrair um padrão que não pode ser interpretado por um ser humano, dada a complexidade do modelo utilizado para representá-lo.
- Conhecimento validado em outras bases de dados ou em amostras de teste, comprovando sua acurácia.
- O padrão descoberto deve estar relacionado com um fenômeno de interesse, ou seja, ter alguma utilidade. Normalmente, em bases de dados com muitas amostras e atributos, é comum encontrar padrões que não representam conhecimento relevante.
- O padrão e o conhecimento descoberto devem apresentar um elemento novo. Em diversos problemas, alguns fenômenos são conhecidos e esperados. Portanto, a descoberta destes não apresenta novidade. Na área médica, por exemplo, diversas relações entre sintomas e diagnósticos já foram mapeadas. A aplicação de técnicas de descoberta de conhecimento deverá buscar novas relações.

Apresentação do Conhecimento

Finalmente, após a conclusão de todas as etapas apresentadas anteriormente, o conhecimento extraído a partir de padrões é apresentado. Uma apresentação apropriada pode ajudar usuários a interpretar com maior facilidade os modelos produzidos pela etapa de mineração de dados, assim como também características dos dados envolvidos no processo (Fayyad et al., 2002). Dentre as diversas formas visuais de apresentação, estão gráficos de funções de duas ou três dimensões, árvores de decisão, grafos, histogramas e uma grande variedade de representações utilizadas na área de visualização de dados estatísticos. A forma de apresentação do conhecimento depende do problema e do modelo utilizado para capturar as propriedades presentes nos dados. Ferramentas interativas, que permitem

que o usuário manipule a representação da informação, podem contribuir principalmente na visualização de dados mais complexos, que podem então ser analisados sob múltiplas perspectivas.

4.3.1 Utilização de Programação Genética

Múltiplas etapas do processo de descoberta de conhecimento apresentam problemas que podem ser resolvidos pela utilização de GP, conforme mostrado nesta seção em alguns exemplos. Em Kalra & Deo (2007), é tratada a presença de dados faltantes, ponto importante da etapa de limpeza de dados. Neste trabalho, GP foi aplicada para a imputação de dados faltantes de uma base de dados com alturas de ondas, em função do tempo, de determinadas regiões da costa da Índia. Esta base de dados, contendo um histórico de quatro anos de coleta de informação, apresenta intervalos de tempo sem amostras, consequência de problemas na coleta de informação (e.g., danificação ou mau funcionamento de instrumentos de telemetria). Neste trabalho, os programas foram representados em forma de árvores, em que os nós não terminais são operações matemáticas e os nós terminais são valores de entrada ou números gerados aleatoriamente. Utilizando amostras desta base de dados, foram criados os conjuntos de treinamento e validação. A função de *fitness* utilizada é baseada no erro quadrático médio (EQM) entre o valor do preditor e o valor esperado, conforme descrito na seção 3.2.3. O programa genético preditor apresentou erro médio de 0.31 metros para as janelas de tempo utilizadas na validação, sendo que as amostras tinham valores de alturas de onda entre 0.5 e 7 metros.

Outra etapa que apresenta problemas que podem ser abordados com GP é a seleção dos dados. Em Bogner & Bouzerdoum (1997), é apresentada a ferramenta *Evolutionary Pre-Processor*, que de forma automática é capaz de extrair características e reduzir a dimensão de bases de dados para indução de classificadores por aprendizado supervisionado. Esta ferramenta é composta por um módulo de GP responsável pelo pré-processamento dos dados e por um conjunto de classificadores. O conjunto de instruções do módulo de GP é composto por operações matemáticas que manipulam os atributos do vetor de entrada. A função de avaliação combina dois elementos: o índice de amostras classificadas incorretamente e uma penalidade em função do tamanho do modelo e do número final de atributos utilizado pelo classificador. O primeiro elemento é semelhante à Equação 3.2, apresentada no capítulo anterior. Porém, ao invés de contabilizar as amostras classificadas incorretamente em número absoluto, é contabilizada em porcentagem. O segundo elemento, apresentado na Equação 4.1, atribui um grau de complexidade a um dado programa i, em que NA é o número de características geradas a partir do vetor de entrada e NN é o número de nós do programa genético. Em Bogner & Bouzerdoum (1997), foram usados os valores $\alpha = 0.01$ e $\beta = 0.000001$. A avaliação final de um dado indivíduo é a soma ponderada da porcentagem de classificação incorreta e do fator de parcimônia. Durante o processo evolucionário, programas candidatos devem minimizar este valor, obtendo classificadores que atendem dois requisitos: performance de classificação e parcimônia.

$$C_i = \alpha * NA + \beta * NN \quad (4.1)$$

4.4 Mineração de Dados

Dentre as etapas do processo de descoberta de conhecimento, a mineração de dados é considerada a etapa mais importante (Han and Kamber, 2006). De certa forma, as etapas que a precedem realizam a preparação dos dados para maximizar a performance desta etapa. Da mesma forma, as etapas que a sucedem são responsáveis por auxiliar a interpretação dos resultados obtidos nesta etapa. Mineração de dados pode ser definida como o emprego de técnicas para encontrar e analisar padrões presentes em bases de dados (Witten and Frank, 2005). Técnicas de classificação de padrões (Duda et al., 2000), análise de agrupamentos (Anderberg, 1973), análise de grafos (Cook and Holder, 2007), dentre outras, podem ser aplicadas para modelar padrões presentes em bases de dados.

A flexibilidade de GP permite modelar soluções para mineração de dados utilizando diversas técnicas. Em Bhattacharya et al. (2001), programação genética linear é aplicada à predição de consumo de energia. Neste trabalho, foi utilizada a representação de programas em código de máquina, em que programas são executados sem a necessidade de um interpretador. Foi utilizada uma base de dados com o consumo de energia em intervalos de meia hora, ao longo de 11 meses, e dados relacionados à temperatura ambiente, fator que influencia diretamente o consumo de energia. Essa base de dados foi manipulada para dar origem a 6 atributos de entrada: temperaturas mínima e máxima, consumo de energia do dia anterior, consumo de período de meia hora atual, estação do ano e dia da semana. Neste trabalho, os indivíduos foram avaliados calculando a raiz do erro quadrático médio, considerando os valores preditos pelos programas candidatos e os valores esperados, dado um conjunto de dados de treinamento. Apesar de utilizar programas representados em código de máquina, este ambiente de indução para predição de valores é válido para outras representações.

Árvores de classificação modelam o relacionamento entre atributos e classes através da combinação de diversas sentenças condicionais. Quinlan (1993) propôs o *C4.5*, algoritmo amplamente utilizado para problemas de aprendizado de máquina, incluindo mineração de dados, que gera árvores de classificação a partir de bases de dados. Programas genéticos que utilizam representação em árvore e possuem nós não terminais para representar conectivos lógicos (*AND, OR, NOT*) e operadores relacionais ($<$, $\leq$, $=$, $>$, $\geq$), e nós terminais para representar valores de entrada e classe de saída, podem ser interpretados como árvores de decisão. Neste tipo de problema, utilizando dados de treinamento rotulados, indivíduos são avaliados baseados no número de classificações corretas e falsos positivos. Em De Falco et al. (2002), ao invés de usar nós terminais indicando a escolha de uma classe, ou seja, uma única árvore de classificação para mapear todas as classes, árvores de classi-

ficação distintas são induzidas para cada classe. A eficiência do algoritmo *C4.5* influenciou trabalhos na área de árvores classificadoras em forma de programas genéticos, como é o caso de Eggermont et al. (2004) em que é descrito um método derivado deste algoritmo *C4.5* para indução de programas genéticos com fenótipo equivalente a árvores de decisão.

4.5 Consultas em Bancos de Dados

4.5.1 *MASSON*

Ryu e Eick (1996b; 1996a) empregaram programação genética para encontrar associações entre objetos e gerar consultas em banco de dados. Para este propósito, foi utilizada a ferramenta *MASSON* que combina um banco de dados orientado a objetos com um ambiente de indução de programas genéticos. A arquitetura de *MASSON* é mostrada na Figura 4.4. O usuário informa conhecimentos do domínio do problema, o esquema do banco de dados orientado a objetos e casos de avaliação. Programas induzidos pelo motor de programação genética utilizam uma interface para realizar operações no banco de dados. Para avaliar candidatos, este mesmo módulo compara o resultado de consultas de programas candidatos com o resultado desejado, especificado nos casos de avaliação providos pelo usuário. Após o processo de indução, *MASSON* provê como resultado consultas de banco de dados orientado a objetos.

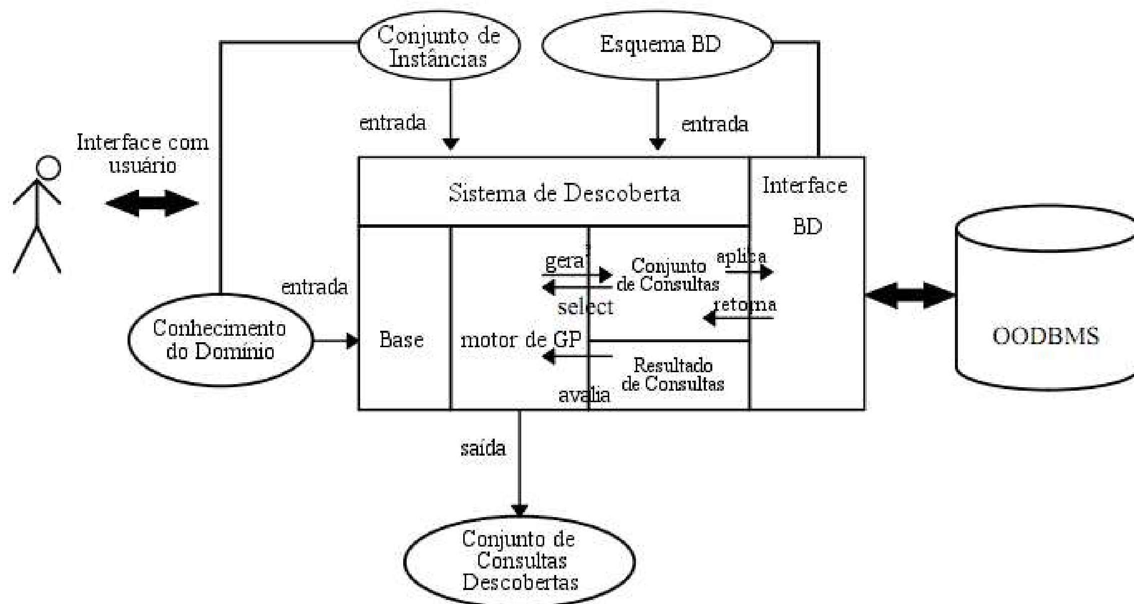

Fig. 4.4: Arquitetura da ferramenta *MASSON*, adaptado de Ryu & Eick (1996a).

No *MASSON*, programas genéticos são representados em forma de árvore. Dada a especifici-

dade desta ferramenta, um conjunto de instruções específico para consulta e manipulação de dados é utilizado, conforme listado abaixo:

- **(SELECT Classe-1 [(Predicado)])**: seleciona todos os objetos da classe *Classe-1* que satisfazem o *Predicado*. Se não houver nenhum predicado, esta instrução seleciona todos os objetos.

- **(RESTRICTED Classe-1 Predicado)**: restringe os objetos de um dado conjunto aos que estão relacionados com uma outra classe *Classe-1* e que satisfazem o *Predicado*.

- **(RELATED Classe-1 Relacionamento-r Classe-2)**: seleciona todos os objeto da *Classe-1* que estão relacionados com objetos da *Classe-2*, dado o *Relacionamento-r*.

- **GET-RELATED Classe-1 Relacionamento-r Classe-2**: operação inversa de **RELATED**. Seleciona todos os objeto da *Classe-2* que estão relacionados com objetos da *Classe-1*, dado o *Relacionamento-r*.

Para mostrar a ferramenta em funcionamento, foi considerada a geração de consultas para um banco de dados com informações pessoais. O esquema deste banco de dados é mostrado na Figura 4.5. Este banco de dados contém as classes *Chamada-telefônica*, *Telefone*, *Pessoa*, *Conta-bancária*, *Compra*, *Transferência* e *transação*. Além das classes, este esquema especifica os atributos e os relacionamentos suportados por cada classe. A classe *Pessoa*, por exemplo, tem os atributos *ssn*, *nome*, *endereço* e *idade*, e está relaciona com a classe *Compras* através do relacionamento *comprou-em/comprado por*.

No processo de avaliação de consultas em forma de programas genéticos, a distância entre uma consulta i e o resultado desejado é calculada pela Equação 4.2, em que T é a cardinalidade do conjunto contendo o resultado desejado, h_i é o número de registros corretos relativos à consulta i e n_i é a cardinalidade do resultado da consulta i. Desta forma, no caso em que a consulta i lista todos os registros corretamente, $D(i) = 0$. No caso da existência de falsos positivos ou falsos negativos, a distância é um valor positivo que cresce em função do número de registros destes tipos.

$$D(i) = T - (h_i * h_i)/n_i; \tag{4.2}$$

Dada a consulta **Q3** ("Listar pessoas que gastaram mais que $3000 em dinheiro em uma loja"), *MASSON* retorna o resultado apresentado abaixo. Este e outros resultados apresentados nestes trabalhos indicam que *MASSON* consegue gerar programas com boa interpretabilidade, capazes de consultar dados organizados num banco de dados orientado a objetos, no intuito encontrar associações entre registros e induzir consultas.

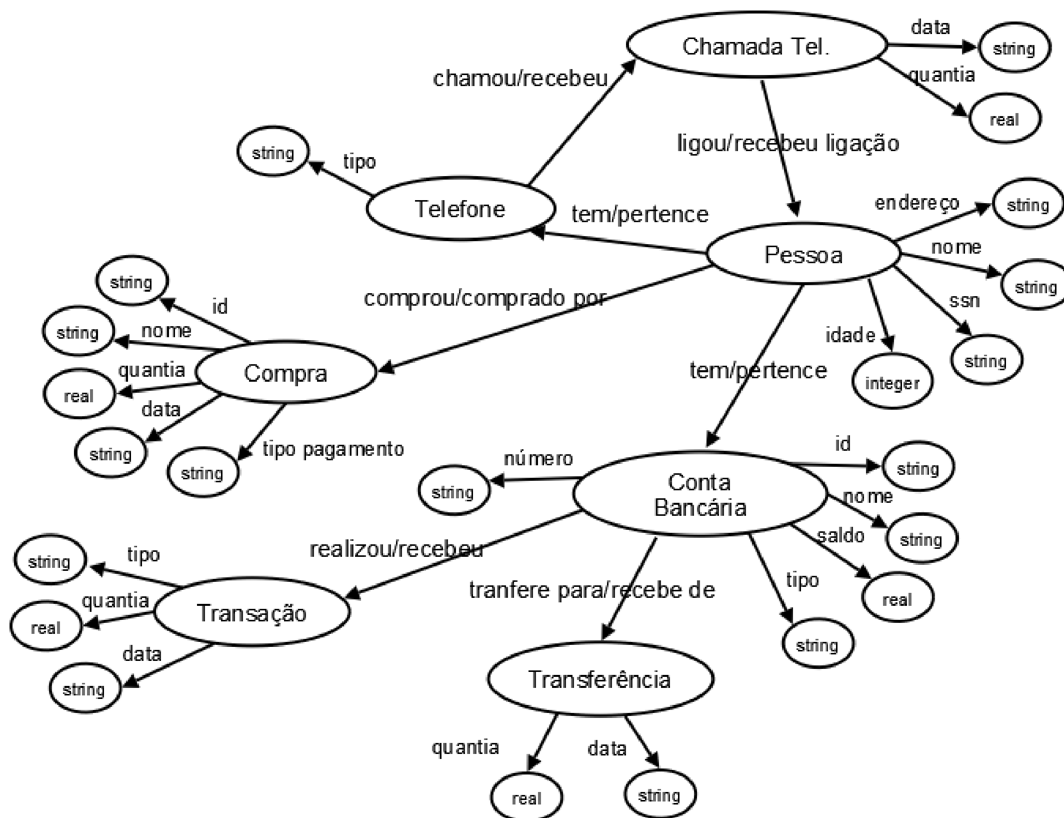

Fig. 4.5: Esquema do banco de dados orientado a objetos, adaptado de Ryu & Eick (1996a).

```
(RELATED person shopped-at
   (SELECT purchase (AND (> amount-spent 3000)
                         (= payment-type 1))))
```

4.5.2 Bancos de Dados Relacionais

A ferramenta *MASSON*, apresentada na seção anterior, é capaz de gerar consultas, modelando relacionamentos entre entidades armazenadas num banco de dados. Entretanto, esta ferramenta utiliza uma arquitetura específica que combina um banco de dados orientado a objetos e uma base de dados de conhecimento do domínio. A utilização de *MASSON* implica no uso de um banco de dados orientado a objetos, pouco usado se comparado com bancos de dados relacionais, e questões específicas de sua arquitetura. Visto que bancos de dados relacionais são predominantes em aplicações de mundo real, abordagens capazes de gerar programas para manipular esse tipo de banco de dados são desejáveis.

Freitas (1997) foi um dos pioneiros na área de geração de consultas de bancos de dados relacionais utilizando GP. Naquele trabalho, o fenótipo dos indivíduos é construído utilizando o *template*

apresentado abaixo:

```
Select Atributo-alvo, Count(*)
From   Relação
Where  descritor-conjunto-tupla
Group  By Atributo-alvo
```

O *atributo-alvo* e a *relação* estão especificados nos dados de treinamento. Programas genéticos são evoluídos para substituir o *descritor-conjunto-tupla* que modela relacionamento de colunas no banco de dados, utilizando as instruções que representam os conectivos lógicos {AND, OR, NOT} e os operadores de comparação $\{<, \leq, =, >, \geq, \neq\}$. Portanto, o fenótipo do indivíduo é uma cláusula SQL que combina o *template* com um programa genético. No caso de um problema de classificação, o *atributo-alvo* é a classe da amostra e o objetivo da abordagem é encontrar a consulta que modela corretamente o padrão de valores das colunas do banco de dados, para cada classe da amostra. Para esta tarefa, foi utilizado um treinamento supervisionado em que o *fitness* de um indivíduo é calculado em função do número de classificações corretas e incorretas, igual a outros trabalhos já discutidos neste capítulo. Apesar das limitações, por exemplo, *relação* de tabelas no banco de dados previamente fixada na consulta e retorno de consulta com apenas um atributo, este trabalho introduziu a possibilidade de evoluir programas capazes de interagir com bancos de dados convencionais.

Salim e Yao (2002) evoluiram consultas SQL para a modelagem soluções para três problemas de classificação de padrões. Assim como no trabalho de Freitas (1997), foi utilizado um *template* de consulta, neste caso, no seguinte formato:

```
SELECT * FROM [nome_tabela] WHERE
```

O genótipo dos indivíduos determina quais atributos, valores e operadores serão utilizados pela cláusula **WHERE**. Portanto, o fenótipo do indivíduo é um trecho de comando SQL que concatenado com o *template* forma uma consulta SQL. Esta consulta é realizada num banco de dados relacional e o resultado é analisado, considerando o número de falsos positivos e falsos negativos, para determinar o *fitness* do indivíduo. Assim como no trabalho de Freitas (1997), a abordagem tem a limitação de operar numa tabela pré-definida. Sendo assim, antes da indução das consultas é necessário criar uma tabela com todos os atributos que serão analisados - etapa existente em alguns processos de DW. Esta limitação dificulta a atuação num banco de dados com múltiplas entidades, distribuídas em tabelas distintas, e com padrões envolvendo entidades e associações, semelhante ao caso apresentado na Subseção 4.5.1.

4.6 Considerações Finais

Este capítulo mostrou que, nas duas últimas décadas, programação genética foi utilizada para buscar a solução de uma grande gama de problemas, obtendo soluções competitivas e, em alguns casos, tão competentes quanto soluções criadas por um especialista humano. Dada a flexibilidade dos algoritmos genéticos, que têm a programação genética como uma especialização, soluções podem ser concebidas para os mais variados problemas, de síntese de circuitos elétricos à programação de robôs autônomos.

Entretanto, mesmo com essa grande gama de aplicações, abordagens de programação genética ainda não têm grande penetração na área de desenvolvimento de sistemas de tecnologia da informação (TI), tanto que é um termo pouco conhecido por engenheiros de software e programadores. Apesar de haver algumas aplicações nas áreas de engenharia de software baseada em busca, conforme mostrado na Seção 2.3, e mineração de dados, algoritmos para modelar processos de sistemas de TI continuam sendo realizados manualmente. Colabora para tal fenômeno o fato de que, quase exclusivamente, pesquisas na área de manipulação de dados, usando programas genéticos, atuam em bases de dados estáticas, principalmente para descoberta de conhecimento e mineração de dados. Uma vez que a modelagem de processos de TI implica numa base de dados dinâmica em que os registros e seus valores variam dependendo do processo em questão, uma nova metodologia precisa ser proposta, voltada para problemas deste tipo.

O capítulo seguinte irá tratar de uma abordagem que se apresenta como um ponto de partida para a geração automática de algoritmos dedicados à modelagem de processos de manipulação de informação em sistemas de TI.

Capítulo 5

Modelagem de Processos de Manipulação de Informação

O Capítulo 2 analisa a evolução do processo de desenvolvimento de software com enfoque nas principais metodologias, ferramentas e princípios. Aquele capítulo também aborda o surgimento de ferramentas e técnicas sofisticadas para o auxílio à tomada de decisão em projetos de software, baseadas no emprego de técnicas de otimização matemática. Outra questão abordada é a automatização de processos envolvidos nas etapas de desenvolvimento de um produto de software. Porém, a etapa que talvez demande maior esforço, a elaboração e codificação de algoritmos, é aquela que apresenta o menor grau de automatização. Dentre as técnicas para geração automática de programas, GP é aquela que apresenta a maior flexibilidade na representação de soluções, uma vez que no uso desta técnica tanto os parâmetros como a estrutura dos programas são elementos controlados pelo processo evolucionário. O Capítulo 3 analisou GP, destacando as diferentes formas de representação de soluções e as etapas do processo evolucionário. Por sua vez, o Capítulo 4 aborda GP do ponto de vista prático, analisando aplicações de GP em diversos campos de atuação.

Apesar da aplicação de GP em diversas áreas, conforme mostrado no capítulo anterior, a geração automática de programas para sistemas de TI é algo pouco explorado. Os trabalhos nesta linha têm focado principalmente na modelagem de padrões entre registros do sistema para aplicações de sistemas inteligentes em TI, por exemplo, sistemas de recomendação, inteligência empresarial, análise de riscos, dentre outras. Porém, este tipo de funcionalidade está presente num nicho específico de aplicações. Portanto, a geração automática de algoritmos ainda não está presente na maioria dos projetos de software em TI.

TI trata do uso da tecnologia para aquisição e processamento de informação para suportar atividades humanas. Por sua vez, um sistema de TI pode ser definido como uma complexa organização de hardware, software, processos, dados e pessoas desenvolvido para auxiliar a execução de tarefas

por indivíduos ou um grupo dentro de uma organização (March and Smith, 1995). Dentro desta definição, no quesito software, se enquadra toda aplicação utilizada para gestão de atividades humanas relacionadas com o processamento de informação, desde um sistema de gerenciamento de pequenos estabelecimentos comerciais a um complexo sistema de gestão de operações financeiras. A popularização deste tipo de sistema, atualmente presente em praticamente todos os campos de atuação, demanda um enorme esforço de desenvolvimento de produtos de software. Este fenômeno pode ser constatado nos números de empresas, ferramentas e tecnologias para este propósito. Sendo assim, a automatização de etapas no processo de desenvolvimento deste tipo de solução gera impactos significativos na qualidade e custo de gerenciamento de informação de diversas atividades humanas.

Neste capítulo é proposta uma nova abordagem para a geração automática de programas, com enfoque num processo mais genérico, a manipulação de informação. Por meio de consulta, inserção, exclusão e atualização de registros armazenados em banco de dados, sistemas de TI modelam processos e gerenciam toda informação relacionada a uma determinada atividade. No caso de um sistema para controle de estoque, operações devem permitir a adição e remoção de itens do estoque de forma sincronizada com a demanda. Neste caso, no banco de dados são armazenados o estado do estoque e informações do sistema. O estado do estoque é composto por informações de cada item presente no mesmo, por exemplo, código, número de unidades, localização física, etc. Por sua vez, informações de usuários autorizados a manipular o estoque, cadastro de fornecedores, regras de negócio fazem parte das informações do sistema. Cada operação que afeta o estado do banco de dados pode ser vista como um processo de manipulação de informação. A partir de uma entrada de dados do usuário, ou de forma automática, um algoritmo que modela a operação de recebimento de novas unidades de um determinado produto, por exemplo, atualiza o número de unidades do produto em questão e registra no sistema que x unidades do produtos p foram entregues pelo fornecedor f na data d. Esta última informação pode ser visualizada em relatórios ou utilizada pelo próprio sistema para melhorar o controle do estoque. Portanto, neste contexto, denomina-se processo de manipulação de informação qualquer processo de manipulação de dados, armazenados num banco de dados, que modela uma operação relacionada a uma atividade.

Algoritmos para manipulação de informação em banco de dados estão presentes na maioria dos sistemas de TI. Entretanto, mesmo com o uso de ferramentas de ORM e *frameworks*, conforme descritas nas Seções 2.2.1 e 2.2.3, procedimentos simples de manipulação de dados ainda são implementados manualmente. A abordagem apresentada neste capítulo não tem como objetivo gerar soluções para qualquer manipulação de dados, mas sim para manipulações mais simples, permitindo assim que programadores humanos se dediquem à execução de tarefas mais complexas.

5.1 A Ferramenta LGPDB

A indução de programas genéticos capazes de manipular dados armazenados num banco de dados relacional exigiu a implementação de uma ferramenta específica para este propósito, denominada LGPDB (do inglês, *Linear Genetic Programming for Databases*). No Capítulo 4 foram analisadas diversas aplicações de GP. Dentre as citadas, algumas apresentam relação com o problema que será abordado pela proposta deste trabalho. No processo de descoberta de conhecimento, conforme descrito na Seção 4.3, etapas fundamentais de preparação dos dados precedem o processo de descoberta de padrões na base de dados. No caso deste trabalho, os dados nos bancos de dados já estão preparados para a etapa de modelagem de padrões, uma vez que os registros foram inseridos manualmente através de uma interface que realiza validações. Mecanismo similar é utilizado praticamente em todos os sistemas de TI para auxiliar os usuários a inserirem os registros corretamente. Mesmo com esse recurso, a ocorrência de algumas falhas é inevitável, porém acontecem numa escala muito menor. No caso de dados relacionados aos processos de sistemas de TI, é comum que os próprios usuários detectem tais falhas durante a utilização do sistema. No caso deste tipo de sistema, o banco de dados é utilizado para organizar informações com propriedades conhecidas, portanto, inconsistências ficam mais evidentes. No caso de KDD, como o conhecimento é algo inacessível *a priori*, falhas e inconsistências podem estar camufladas nos dados.

Conforme mostrado na Subseção 4.5.2 do Capítulo anterior, Freitas (1997), Salim e Yao (2002) desenvolveram técnicas para induzir consultas em bancos de dados relacionais para abordar problemas de classificação e *data mining*. Entretanto, neste dois trabalhos, as regras de filtragem são geradas para atuar num conjunto de dados organizado numa única estrutura de dados. Portanto, em aplicações práticas, é necessário criar essa estrutura, por exemplo, uma tabela de banco de dados, contendo todos os atributos que serão analisados. Esta limitação impossibilita a utilização destas abordagens para a indução de consultas em bancos de dados em que a informação está distribuída em múltiplas tabelas, como é o caso de praticamente todo sistema de TI. Por outro lado, conforme mostrado na Subseção 4.5.1, Ryu e Eick (1996b; 1996a) desenvolveram uma ferramenta para geração de consultas em bancos de dados orientado a objetos. Neste caso, os dados estão distribuídos em múltiplas estruturas de dados, semelhante aos bancos de dados utilizados em sistemas de TI. Porém, neste caso, a limitação é a utilização de uma modalidade de banco de dados dificilmente usada no desenvolvimento de sistemas de TI. Portanto, LGPDB deverá permitir a atuação em banco de dados relacional com atributos de entidades distribuídos em múltiplas tabelas. Além do mais, para a modelagem de processos de manipulação de informação é necessário gerar operações que alterem os dados no banco de dados, não apenas consultas.

Em relação a arquitetura, representação dos dados e método de aprendizado, o LGPDB foi inspirado em alguns elementos da ferramenta *MASSON*, proposta por Ryu e Eick (1996b; 1996a). Em

ambas os dados são representados de forma estruturada, seja utilizando objetos ou entidades e relacionamentos. Representações deste tipo permitem armazenar dados referentes a múltiplos elementos de uma atividade numa única base de dados. Objetos ou entidades podem ter relacionamentos entre si, permitindo associar registros de diferentes elementos de uma dada atividade. No caso do *MASSON*, como já citado, foi utilizado um banco de dados orientado a objetos com a modelagem mostrada na Figura 4.5. Esta representação tem como vantagem a organização e manipulação dos dados utilizando um formato mais próximo do mundo real. Funcionalidades como herança entre classes, encapsulamento, sobrecarga de operações permitem a representação de bancos de dados de maior complexidade, de forma mais coesa. Apesar destas qualidades, este modelo de banco de dados ainda apresenta pouca adesão no segmento de aplicações de TI. O modelo relacional, utilizado pelas principais soluções de bancos de dados e dominante no segmento de TI, é mais simples, rápido e apresenta um legado de décadas de utilização. Este último formato foi escolhido para ser utilizado por LGPDB considerando que a representação dos dados no mesmo formato da maioria dos sistemas existentes no mercado aumenta a viabilidade de aplicação desta ferramenta em problemas do mundo real.

Os elementos que compõem LGPDB são mostrados na Figura 5.1. O módulo de indução de programas é responsável pela evolução de programas de computador por meio de programação genética. Soluções candidatas são executadas no ambiente de execução de programas que tem acesso ao banco de dados de indução. O usuário especifica o problema por meio dos casos de avaliação que são executados no banco de dados de validação. Comparando o resultado provido por soluções candidatas e aquele especificado nos casos de avaliação, o módulo de indução de programas é capaz de avaliar indivíduos e direcionar a busca por soluções melhores. Os bancos de dados de indução e validação são cópias do banco de dados original, especificado pelo usuário. A utilização de duas instâncias de bancos de dados facilita o gerenciamento de múltiplos estados de bancos de dados. Em alguns casos, programas podem manipular registros em múltiplas tabelas. Utilizando instâncias distintas, independente do número de tabelas manipuladas, o processo de comparação de estados de bancos de dados precisa apenas comparar as tabelas e registros das duas instâncias para mapear a diferença entre os dois estados de bancos de dados. O mesmo processo poderia ser feito utilizando apenas uma instância de banco de dados, porém seria necessário um método mais sofisticado para gerenciar em memória diferentes estados de uma mesma instância de banco de dados.

Do ponto de vista de aprendizado, tanto o LGPDB como *MASSON* utilizam um método supervisionado, em que soluções emergem a partir de tentativas se modelar exemplos de processos providos pelo usuário. As principais diferenças são o tipo de banco de dados, a forma em que soluções são representadas e a capacidade de alteração de registros, esta última sendo provida apenas pelo LGPDB.

Os elementos apresentados na Figura 5.1 fazem parte dos dois principais módulos de LGPDB, o

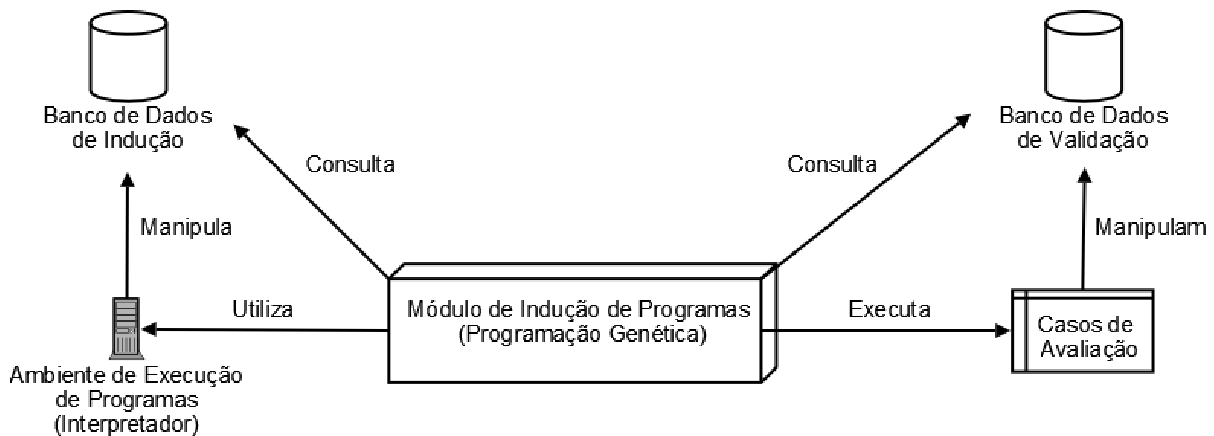

Fig. 5.1: Elementos que compõem a arquitetura da ferramenta LGPDB.

módulo de indução de programas e o sistema de gerenciamento de banco de dados (SGBD), que são analisados em maior profundidade nas próximas seções.

5.1.1 Sistema de Gerenciamento de Banco de Dados

O LGPDB utiliza um sistema de gerenciamento de banco de dados (SGBD) exclusivo, desenvolvido para atender as necessidades específicas da ferramenta. Se comparado com outros SGBDs, como por exemplo o PostgreSQL ou MySQL, esta implementação provê um número bem menor de funcionalidades, porém algumas foram desenvolvidas especificamente para indução e execução de programas genéticos. Assim como a maioria dos SGBDs, esta implementação suporta a execução de transações, em que múltiplas operações são executadas atomicamente e podem ser revertidas se desejado ou em caso de algum erro (Silberschatz et al., 2002). Este recurso é uma premissa para o processo de indução de programas uma vez que soluções candidatas irão explorar diversas operações no banco de dados que precisam ser revertidas, para dar início a avaliação da próxima solução.

Funcionalidades Específicas para Indução de Programas Genéticos

No processo de evolução de programas, uma etapa fundamental é a avaliação, em que cada programa recebe uma nota, indicando o quão apto está para a resolução do problema alvo. Uma vez que o LGPDB evolui programas para manipulação de um banco de dados relacional, operações de seleção, inserção, exclusão e atualização de registros podem ser executadas diversas vezes durante o processo de avaliação de um único programa. Como será mostrado nas próximas sessões, o LGPDB trabalha com populações de centenas ou até milhares de programas que, consequentemente, implica uma quantidade massiva de operações realizadas no banco de dados. Para acelerar a execução das instruções de manipulação de dados durante o processo de avaliação, utiliza-se um banco de dados

em memória principal, em que todos os registros são manipulados em memória de acesso aleatório (RAM, do inglês *Random Access Memory*). Esta modalidade de banco de dados apresenta melhor performance de acesso aos dados, quando comparada com bancos de dados tradicionais que armazenam os dados em disco rígido, visto que o tempo de acesso à memória principal pelo processador é significantemente menor e invariante em relação ao endereço do dado (Garcia-Molina and Salem, 1992).

Além da questão de performance, o módulo SGBD fornece as seguintes funcionalidades específicas para a indução de programas de computador: armazenamento de registros sem tipagem de dados e a possibilidade de determinar a diferença entre duas instâncias de banco de dados em memória.

Para simplificar as operações que manipulam dados no bancos de dados e diminuir a quantidade de elementos que devem ser induzidos no processo evolutivo, uma funcionalidade suportada pelo SGBD implementado é o armazenamento sem tipagem de dados. Desta forma, todas as informações são armazenadas num único formato e os tipos são determinados em tempo de execução, dado o conteúdo das variáveis. Conversões entre formatos, por exemplo, para realização de operações matemáticas, no caso de dados numéricos, ou comparação entre sequências de caracteres, no caso de dados textuais, são feitas dinamicamente e de forma automática.

No processo de indução de programas são utilizados duas instâncias de banco de dados em memória, uma utilizada pelos programas que representam soluções candidatas e outra utilizada para validação. Dado o estado inicial do banco de dados de indução, o objetivo dos programas é encontrar a sequência de operações que leva este banco de dados ao mesmo estado do banco de dados de validação. Portanto, uma funcionalidade que o SGBD fornece é a capacidade de realizar a diferença entre duas instâncias de bancos de dados. Essa funcionalidade tem duas aplicações no processo de indução:

1. **Listar registros que devem ser manipulados:** comparando o estado inicial do banco de dados de indução com o resultado esperado após a manipulação, contido na base de dados de validação, é possível listar os registros que devem ser manipulados. A manipulação desses registros por soluções candidatas pode ser incentivada no processo de indução, por exemplo, atribuindo maior peso a esses registros no processo de avaliação.

2. **Comparar os resultados final e desejado:** comparando novamente essas duas bases de dados, porém agora no estado final, é possível avaliar o quão apta está a solução representada por um dado programa. Com base nesta comparação, é determinado o valor de *fitness* desta solução candidata.

Tab. 5.1: Operações suportadas pelo módulo SGBD.

Instrução	**Descrição**
createTable	Dado o nome da tabela e lista de atributos, cria uma nova tabela no banco de dados.
deleteTable	Dado o nome de uma tabela, faz a remoção da mesma do banco de dados.
difference	Dadas duas instâncias do banco de dados em memória, compara as duas instâncias e retorna os registros distintos em cada tabela do banco de dados.
clone	Dada uma instância do banco de dados, cria uma nova instância em memória com as mesmas características, incluindo os registros em cada tabela.
select	Dado o nome da tabela, retorna todos os registros armazenados nesta tabela.
delete	Dado o nome da tabela e valor da chave primária, exclui um registro.
insert	Dado o nome da tabela e os valores dos atributos, insere um novo registro.
update	Dado o nome da tabela, nome do atributo e novo valor, atualiza um registro no banco de dados.
commit	Valida o estado atual do banco de dados.
rollback	Retorna o banco de dados para o último estado válido.

Instruções e API

Programas e usuários manipulam os dados por meio de um conjunto de operações, providas pelo módulo de SGBD, conforme mostrado na Tabela 5.1. As quatro primeiras operações são utilizadas pelos usuários para configurar o banco de dados e o ambiente de indução. As operações **select**, **delete**, **insert** e **update** são utilizadas pelos programas gerados por LGPDB, acessíveis por meio de instruções utilizadas pelos programas genéticos, conforme é descrito na Seção 5.1.2. As instruções **commit** e **rollback** são utilizadas no processo evolucionário para salvar e recuperar estados do banco de dados. Estas duas últimas instruções são úteis no processo de avaliação, em que todos os programas candidatos devem ser avaliados sob as mesmas condições. Portanto, é necessário desfazer as operações de uma dado programa antes da avaliação do próximo.

5.1.2 Módulo de Indução de Programas

O SGBD, apresentado na seção anterior, é responsável por organizar e prover mecanismos de manipulação de dados a usuários e programas. No caso da modelagem de processos de manipulação de informação, o objetivo é encontrar o conjunto correto de instruções e parâmetros para realizar uma

determinada operação no banco de dados. Para tal propósito, programas utilizam um conjunto de instruções para acessar os dados armazenados no banco de dados, assim como para manipular estes dados em memória, por exemplo, para realizar filtragens ou alterar valores. Esta seção apresenta o módulo responsável pela indução de programas genéticos por meio de um método de computação evolucionária, com as mesmas etapas fundamentais descritas na seção 3.2.

Representação de Programa

Em GP, existem diversas formas de representar programas, conforme apresentado no Capítulo 3. O LGPDB utiliza representação linear, portanto, programas são representados como uma sequência de instruções executadas em ordem. Esta representação foi escolhida pois se assemelha com a forma pelo qual programas são representados em linguagens de programação imperativas, por conseguinte, apresenta um maior grau de interpretabilidade para o ser humano se comparada com as representações em árvore ou grafo. Este grau de interpretabilidade foi fundamental na fase de desenvolvimento do LGPDB, visto que programas gerados automaticamente pela ferramenta precisavam ser verificados por um ser humano. Através da realização de testes de mesa foi possível verificar se todos os estados de execução de programas produzidos pela ferramenta estavam condizentes com a definição do ambiente de execução. Além da questão da interpretabilidade, a representação linear também tem como vantagem a complexidade de implementação de operadores genéticos. Uma vez que as instruções operam de forma independente, apenas compartilhando variáveis em memória, os operadores genéticos não precisam considerar questões estruturais do programa. No caso das representações em árvore e grafo, é necessário garantir que o programa esteja estruturalmente consistente, por exemplo, que todos os nós do programa em árvore estejam conectados e que o programa em grafo não entre em laço infinito.

Outra característica da representação de programas utilizada pelo LGPDB é a utilização de instruções de alto nível. No caso da utilização de instruções num nível menor, como nos diversos exemplos apresentados em Brameier & Banzhaf (2007) ou nos programas induzidos utilizando a abordagem FSGP (do inglês, *Function Sequence Genetic Programming*) (Wang et al., 2009), programas são formados basicamente pela combinação de operações matemáticas realizadas em variáveis primitivas em memória. Programas que trabalham com instruções neste nível são capazes de atuar numa gama maior de problemas que envolvem criação de mapeamento entre entrada e saída, uma vez que as instruções são de propósito genérico. Por outro lado, o espaço de soluções é muito maior e, em alguns casos, pode tornar inviável a indução de programas. Mas é importante ressaltar que não existe uma representação melhor que a outra. Essa decisão vai depender do problema que será abordado. Em Brameier & Banzhaf (2001), utilizando instruções deste tipo, são induzidos classificadores para realização de diagnósticos médicos, considerando diferentes bases de dados, que obtiveram poder de

classificação e generalização tão competitivos quanto uma abordagem utilizando redes neurais artificiais do tipo perceptron de múltiplas camadas. Nesse campo de atuação, independentemente do problema representado pela base de dados, o objetivo é combinar operações matemáticas utilizando um vetor de atributos de entrada, no intuito de criar um mapeamento de classificação que gera uma determinada saída, normalmente a classe da amostra.

Entretanto, os problemas abordados com LGPDB apresentam características bem distintas que justificam a utilização de uma representação específica, empregando instruções que operam em alto nível, que são menos comuns na literatura. No caso de LGPDB, ao invés de manipular variáveis primitivas em memória com funções matemáticas, estruturas de dados mais complexas, contendo múltiplas variáveis, são manipuladas por instruções capazes de requisitar registros num banco de dados, realizar filtragens, alterar valores ou relacionar múltiplas estruturas deste tipo. A instrução **select("aluno", rs1)**, por exemplo, quando executada, conecta-se ao banco de dados, requisita os registros armazenados na tabela "aluno", copia os registros requisitados para uma estrutura de dados em memória **rs1**.

Em abordagens para tratar de problemas de KDD que envolvem amostras que descrevem um determinado fenômeno ou objeto, usualmente as amostras são apresentadas, uma a uma, ao modelo que irá representar uma determinada associação. Por outro lado, no caso da indução de programas que manipulam um banco de dados que armazena registros de múltiplas entidades e relacionamentos, na maioria dos casos, são modelados processos envolvendo apenas um subconjunto destes elementos. Do ponto de vista de sistemas de TI, quanto maior o número de entidades e relacionamentos no banco de dados, menor tende a ser a porcentagem destes elementos que participam de um determinado processo. No caso de um banco de dados utilizado para gerir uma universidade, por exemplo, para listar os alunos de um determinado departamento que têm publicações em conjunto com alunos de um outro departamento, não há necessidade de relacionar informações com as tabelas de funcionários, cursos, salários, etc. Portanto, a modelagem de processos neste tipo de banco de dados pode ser divida em duas etapas: (1) determinar quais entidades e relacionamentos fazem parte da solução; (2) modelar as associações de registros destas entidades e relacionamentos.

Abaixo são listadas as instruções fornecidas pelo LGPDB para abordar este tipo de problema:

- **Select(Table tb, ResultSet rs)**: seleciona todos os registros armazenados na tabela **tb** e os armazena no conjunto de resultado **rs**. Antes deste processo ser executado, todo conteúdo armazenado em **rs** é excluído da memória.

- **Filter(ResultSet rs, Attribute attr, Rule r, InputValue v)**: filtra o conjunto de resultado **rs**, aplicando a regra **r** considerando o valor de entrada **v** e o atributo **attr**.

- **Related(ResultSet rs1, ResultSet rs2)**: para todo registro em **rs1**, se não houver uma chave

estrangeira associando-o com um registro em **rs2**, este registro é removido da memória de **rs1**.

- **UnRelated(ResultSet rs1, ResultSet rs2)**: para todo registro em **rs1**, se houver uma chave estrangeira associando-o com um registro em **rs2**, este registro é removido da memória de **rs1**.

- **Delete(Table tb, ResultSet rs)**: exclui todos os registros na tabela **tb** com a mesma chave primária dos registros em **rs**.

- **CreateRelation(Table tb, ResultSet rs1, ResultSet rs2)**: se existir uma chave estrangeira para associação de registros entre as tabelas com dados em **rs1** e **rs2**, criar esta associação e inserir um novo registro na tabela **tb**.

- **SetRelation(ResultSet rs1, ResultSet rs2)**: se existir uma chave estrangeira que possibilita associar registros entre as **rs1** e **rs2**, associar os registros e atualizar a tabela no banco de dados.

Na lista apresentada anteriormente é possível observar que as instruções do LGPDB manipulam parâmetros de tipo específico, ao invés de variáveis que representam qualquer endereço de memória. Esta representação de programa com tipagem de dados (Montana, 1995), em conjunto com operadores genéticos que garantam a parametrização correta das soluções, reduz o espaço de busca de soluções, uma vez que para cada parâmetro, só variáveis do tipo correto podem associadas no genótipo. Os diferentes tipos de variáveis manipuladas por instruções do LGPDB são apresentadas na Tabela 5.2.

Tab. 5.2: Tipos de variáveis manipuladas por instruções de programa do LGPDB.

Tipo	**Valores suportados**
Table	nome de uma tabela do banco de dados.
ResultSet	índice de um determinado conjunto de resultado. O número de conjuntos de resultado disponíveis para o programa é um dos parâmetros de indução do LGPDB.
Rule	=, !=, > e <.
Attribute	nome de qualquer atributo contido num determinado conjunto de resultado.
InputValue	índice de um dos valores de entrada especificado pelo usuário. Este número está associado aos valores de entrada especificados nos casos de teste.

Um exemplo da utilização destas instruções e variáveis é mostrado no programa abaixo. Este programa hipotético lista todos os alunos da Unicamp associados ao Departamento de Engenharia da Computação e Automação Industrial (DCA) da Faculdade de Engenharia Elétrica e de Computação (FEEC).

```
1:   select(departamento,rs2)
2:   filter(sigla,equals,"DCA",rs2)
3:   select(faculdade, rs1)
4:   filter(sigla,equals, "FEEC", rs1)
5:   relate(rs2, rs1)
6:   select(aluno, rs1)
7:   related(rs1,rs2)
```

Na linha **1**, todos os registros da tabela *departamento* são selecionados no banco de dados e armazenados na conjunto de resultado **rs2**. Em seguida, na linha **2**, o conjunto de resultado **rs2** é filtrado e todos os registros tendo a sigla de departamento diferentes de "DCA" são removidos Adiante, na linha **3**, todos os registros da tabela *faculdade* são selecionados e armazenados no conjunto de resultado **rs1**. Na linha **4**, o conjunto de resultado **rs1** é filtrado e todos os registros tendo a sigla diferente de "FEEC" são removidos. Em seguida, na linha **5**, em **rs2** são mantidos apenas os registros que têm uma chave estrangeira que os associa com algum registro em **rs1**. Portanto, em **rs2** são mantidos os departamentos com a sígla "DCA" que estão associados à faculdade de sigla "FEEC". Este relacionamento é necessário pois a sigla de departamento é única apenas numa mesma faculdade, ou seja, pode haver departamentos com a mesma sigla em faculdades distintas, como são os casos do Departamento de Estatística (DE) e Departamento de Enfermagem (DE). Na linha **6**, todos os registros da tabela aluno são selecionados e armazenados no conjunto de resultado **rs1**. Finalmente, na última linha, são relacionados todos os alunos associados ao departamento "DCA" contido na faculdade "FEEC". Para simplificar o exemplo, esta modelagem hipotética permite apenas que o aluno esteja associado a um único departamento.

Esta mesma operação poderia ser realizada com o comando SQL apresentado abaixo:

```
SELECT * FROM alunos, departamento, faculdade
WHERE
faculdade.sigla = ''FEEC''
AND departamento.sigla = ''DCA''
AND departamento.faculdade = faculdade.id
AND aluno.departamento = departamento.id;
```

Como pode ser observado, comparando um programa no formato utilizado pelo LGPDB com um comando SQL equivalente, a instrução **related()** é equivalente a igualar chaves estrangeira e primária de duas tabelas distintas. Já a instrução **filter()**, é equivalente a filtrar registros através de uma condição de comparação entre atributo e valor.

Método de Aprendizado e Avaliação

A evolução de programas por LGPDB utiliza dois elementos fundamentais: um conjunto de exemplos do processo a ser modelado e um método de avaliação. Sendo assim, para cada algoritmo que será gerado automaticamente, é fornecido um conjunto de resultados desejados em função de determinados parâmetros. Este resultado pode ser tanto um conjunto de dados que deve ser consultado como operações que devem ser realizadas no banco de dados de indução. Portanto, o aprendizado em LGPDB acontece de forma supervisionada, uma vez que o usuário provê exemplos do comportamento que deseja modelar com a ferramenta. No caso da modelagem de processos de manipulação de informação, a relação entre os parâmetros e o resultado desejado faz parte da modelagem do sistema, normalmente realizada na etapa de análise de requisitos.

Tendo o resultado desejado e o resultado provido por um programa candidato, uma solução pode ser avaliada em função da distância destes dois resultados. Para cada programa candidato é dada uma nota, denominada grau de adaptação ou *fitness*, dentro do intervalo [0,1], sendo que 1 indica que esta solução é ótima para o caso de avaliação atual e valores inferiores a 1 indicam soluções parciais. Quanto menor o valor de *fitness*, pior é a solução candidata. A Equação 5.1 retorna o valor de *fitness* do programa *p* considerando a medida distancia *D*(p) entre o resultado provido pelo programa e o resultado desejado. M é equivalente a distância de um programa sem instruções, ou seja, determina a distância máxima considerada na avaliação de um indivíduo. No caso de indivíduos que eventualmente possam ter o valor *D(p)* superior a M, indivíduos destrutivos que são piores do que um programa nulo, a distância M é aplicada.

$$F(p) = \frac{(M+1) - D(p)}{M+1} \tag{5.1}$$

Programas induzidos pelo LGPDB podem manipular dados num banco de dados relacional através de quatro operações básicas: seleção, inserção, exclusão e atualização. Para cada uma destas operações existe uma métrica de distancia *D*(p). Portanto, durante o processo de indução e considerando o caso de avaliação atual, o LGPDB determina qual tipo de operação está sendo realizada numa dada tabela para determinar qual das métricas de distância será considerada. No caso de operações de seleção, a métrica é apresentada na Equação 5.2 em que o grau de adaptação de um programa *p* é calculado somando o número de falsos negativos *FN* e falsos positivos *FP* para cada um dos *E* casos de avaliação. A constante α é utilizada para penalizar os falsos negativos, induzindo os primeiros programas a consultarem um conjunto contendo todos os registros desejados e diversos registros não desejados. A partir deste estágio, o objetivo será remover os registros indesejados por meio das instruções de filtragem.

$$D_{query}(p) = \sum_{i=0}^{E}(FN_i * \alpha + FP_i) \tag{5.2}$$

A operação de exclusão, de certa forma, pode ser vista como uma operação de seleção. Afinal, um dos parâmetros da instrução **Delete()** é um **ResultSet** contendo os registros que devem ser excluídos. Sendo assim, se os registros corretos forem selecionados e armazenados no **ResultSet** utilizado pela instrução **Delete()** de um programa que modela um processo de exclusão, esta operação será realizada de forma correta. Partindo deste raciocínio, a métrica de distância para operações de exclusão é muito semelhante à utilizada para operações de seleção, passando apenas o peso aplicado por α aos falsos positivos, conforme mostra a Equação 5.3. Desta forma, análogo ao processo de indução de programas para seleção, nas primeiras gerações os programas candidatos tendem a excluir todos os registros desejados e diversos registros não desejados. Ao longo das gerações, através de filtragens, os registros que não devem ser excluídos são removidos do **ResultSet** utilizado como parâmetro da instrução **Delete()**. Ao final do processo, espera-se que neste **ResultSet** estejam apenas os registros que devem ser excluídos.

$$D_{delete}(p) = \sum_{i=0}^{E}(FN_i + FP_i * \alpha); \tag{5.3}$$

No caso de consulta ou exclusão, a operação é realizada em nível de registro, portanto, todos os atributos estão envolvidos obrigatoriamente. Consultas retornam registros completos e exclusões removem registros incluindo todos os seus atributos. Entretanto, no caso de inserção ou atualização, operações são realizadas em nível de atributo. Sendo assim, em muitos casos, registros não são adicionados ou atualizados por completo numa única operação. No caso de uma inserção, por exemplo, uma instrução pode criar um registro inicial e outra ajustar os valores de atributos. No caso de uma atualização, normalmente apenas um subconjunto dos atributos serão manipulados. Portanto, no caso destes dois tipos de manipulações, a função que determina a distância deve considerar também a diferença entre atributos de registros. Para um determinado caso de avaliação, numa primeira etapa, é determinado o conjunto de registros que deve ser manipulado, por meio da comparação dos estados inicial e final do banco de dados de validação. A função de distância considera dois elementos: se o programa manipulou os registros corretos e a distância de *Hamming* (DH) entre os registros manipulados. Se dois registros têm os atributos iguais, DH é zero. Caso contrário, DH é o número de atributos distintos. A distancia total entre dois resultados é o somatório das DH entre cada par de registro comparado. No caso de operações de atualização, registros marcados para serem alterados têm o valor da DH multiplicado por um α para punir ainda mais os programas que os desconsideraram.

Detecção de Introns

Na Seção 3.2.5 foi discutida a importância de se detectar e não considerar os introns - instruções que não afetam o resultado do programa perante o método de avaliação - durante o processo de indução, com o objetivo de reduzir o custo computacional e, consequentemente, diminuir o tempo necessário para a geração automática de programas. Além disso, programas sem introns têm uma interpretabilidade muito maior e facilitam as análises para validação de resultados.

Bramier e Banzhaf (2001) propuseram um método para detectar introns estruturais que são instruções que manipulam variáveis, num dado trecho do programa, que não afetam o resultado. Na Figura 5.2 é mostrado um exemplo de programa no formato do LGPDB com introns estruturais, destacados em negrito. Na linha 1, o comando **select** é um intron pois altera a variável **rs1** que é reinicializada na linha seguinte. Na linha 4, o comando **filter** é um intron pois filtra a variável **rs4** que não foi inicializada, portanto, não afeta o resultado da execução do programa.

1:	**select(livro, rs1)**
2:	select(livro, rs2)
3:	select(editora, rs3)
4:	**filter(nome, equals, x, rs4)**
5:	filter(nome, equals, x, rs3)
6:	relate(rs2,rs3)

Fig. 5.2: Exemplo de programa no formato de LGPDB. Em negrito estão destacadas instruções que são introns.

O Algoritmo 1 apresenta uma versão adaptada para o formato do LGPDB do algoritmo de detecção de intron proposto por Bramier e Banzhaf. Basicamente, este algoritmo navega pelas instruções do programa em ordem contrária. Partindo da última instrução, analisando a manipulação das variáveis que estão vinculadas a saída do programa. No caso do LGPDB, está definido *a priori* que resultados de consultas devem ser armazenados na variável **rs1**, portanto, na linha 1, esta variável é adicionada na lista $infVars$ que contém as variáveis que afetam o resultado de execução do programa. Na linha 3 é iniciado o processo que percorre o programa em ordem contrária. Se uma instrução inicializa ou modifica uma variável presente na lista $infVars$, esta instrução não é um intron. Caso contrário, esta instrução é um intron pois manipula variáveis que no trecho atual do programa não afetam o resultado. Se uma instrução que não é um intron tem outros **ResultSet** como parâmetro, estes são adicionados a lista $infVars$ pois estão relacionados com um **ResultSet** que afeta o resultado do programa. No caso da instrução **relate(ResultSet rs1, ResultSet rs2)**, por exemplo, se o **ResultSet** $r1$ está em $infVars$, consequentemente o **ResultSet** $r2$ também afeta o resultado do programa, uma vez que o conteúdo deste **ResultSet** irá determinar o novo valor de $r1$, dado o relacionamento entre os registros armazenados nestas variáveis. Este processo é realizado até se chegar à

primeira instrução do programa. Instruções marcadas como introns não precisam ser executadas no processo de avaliação e podem ser removidas ao final do processo de indução.

Algoritmo 1 Detecção de Intron

Entrada: Program $prog$, OperationType $opType$

1: $infVars \leftarrow newList()$
2: add $rs1$ to $infVars$
3: **for** $i = prog.size - 1; i >= 0; i - -$ **do**
4: $influenceOutput \Leftarrow false$
5: $inst \leftarrow prog.getInstruction(i)$
6: $pVars \leftarrow getParameterList(inst)$
7: $rVars \leftarrow getRenewedParameterList(inst)$
8: $mVars \leftarrow getModifiedParameterList(inst)$
9: **for** $j = 0$ **to** $rVars.size$ **do**
10: $var \leftarrow rVars.get(j)$
11: **if** $infVars$ contains var **then**
12: remove var from $infVars$
13: $influenceOutput \leftarrow true$
14: **end if**
15: **end for**
16: **for** $j = 0$ **to** $mVars.size$ **do**
17: $var \leftarrow mVars.get(j)$
18: **if** $infVars$ contains var **then**
19: $influenceOutput \leftarrow true$
20: **end if**
21: **end for**
22: **if** $influenceOutput$ is true **then**
23: **for** $j = 0$ **to** $pVars.size$ **do**
24: $var \leftarrow pVars.get(j)$
25: **if** $rVars$ not contains var **then**
26: **if** $infVars$ not contains var **then**
27: add var to $infVars$
28: **end if**
29: **end if**
30: **end for**
31: **else**
32: mark $inst$ as intron
33: **end if**
34: **end for**

No caso do LGPDB, este algoritmo só pode ser aplicado na indução de programas para realizarem consultas no banco de dados pois este é o único tipo de funcionalidade que tem o resultado armazenado numa variável de saída (**rs1**). No caso das funcionalidades para inserção, exclusão e adição, um

novo método precisa ser implementado, capaz de determinar se uma instrução irá alterar registros que não devem ser manipulados.

Entretanto, no caso deste trabalho, o principal objetivo da detecção de intron não é acelerar o processo evolucionário pela redução do custo computacional, mas sim remover este tipo de intron no final do processo. Fornecer programas sem instruções desnecessárias, além de facilitar o uso da abordagem na prática, também facilita a interpretação dos resultados, por exemplo, para um trabalho científico em que não basta mostrar que o programa modela a relação de entrada e saída com precisão. Neste tipo de trabalho é importante analisar e entender como o programa faz isso, portanto, modelos mais interpretáveis podem permitir análises mais aprofundadas, principalmente com relação à qualidade das soluções.

Para permitir a remoção de introns ao final do processo mesmo no caso de funcionalidades que alteram o estado do banco de dados, um algoritmo simples, porém custoso, foi implementado. A solução é custosa pois é necessário executar e avaliar o programa múltiplas vezes para remover os introns. O princípio é simples, partindo da primeira instrução e através de um processo iterativo, cada instrução de um programa p é desconsiderada da execução, gerando um novo programa p' que é executado e avaliado. Se a avaliação do programa p' permanece igual a do programa p, a instrução desconsiderada na execução é removida e o processo é repetido desde o começo no programa p'. O algoritmo termina após chegar numa versão do programa em que qualquer instrução que seja removida, afeta a avaliação do programa. A utilização desta abordagem é viável pois é aplicada apenas à solução ótima, obtida ao final do processo evolucionário.

Processo Evolucionário

O processo evolucionário adotado pelo LGPDB, combinando todas as etapas apresentadas anteriormente, é muito semelhante ao apresentado na Seção 3.2. Basicamente, a única diferença está no gerenciamento dos bancos de dados. Diferentemente das aplicações de programação genética para KDD em que o banco de dados permanece estático, no caso do LGPDB, é necessário voltar os bancos de dados de indução e validação para os estados iniciais, após a avaliação de cada indivíduo. Na Figura 5.3 é mostrado o fluxograma do processo evolutivo do LGPDB.

Antes de iniciar o processo de evolução de programas, são criadas duas instâncias do banco de dados, uma para indução e outra para validação. Estas instâncias são cópias do banco de dados configurado e preenchido previamente pelo usuário. Durante o processo evolucionário, os programas candidatos operam no banco de dados de indução e os casos de avaliação, no caso de funcionalidades que alteram os dados do banco de dados, são aplicados no banco de dados de validação. Conforme descrito na seção que trata do método de avaliação, uma comparação entre estes dois bancos de dados determina o grau de aptidão de um dado indivíduo (*fitness*). Em seguida, os estados dos bancos de

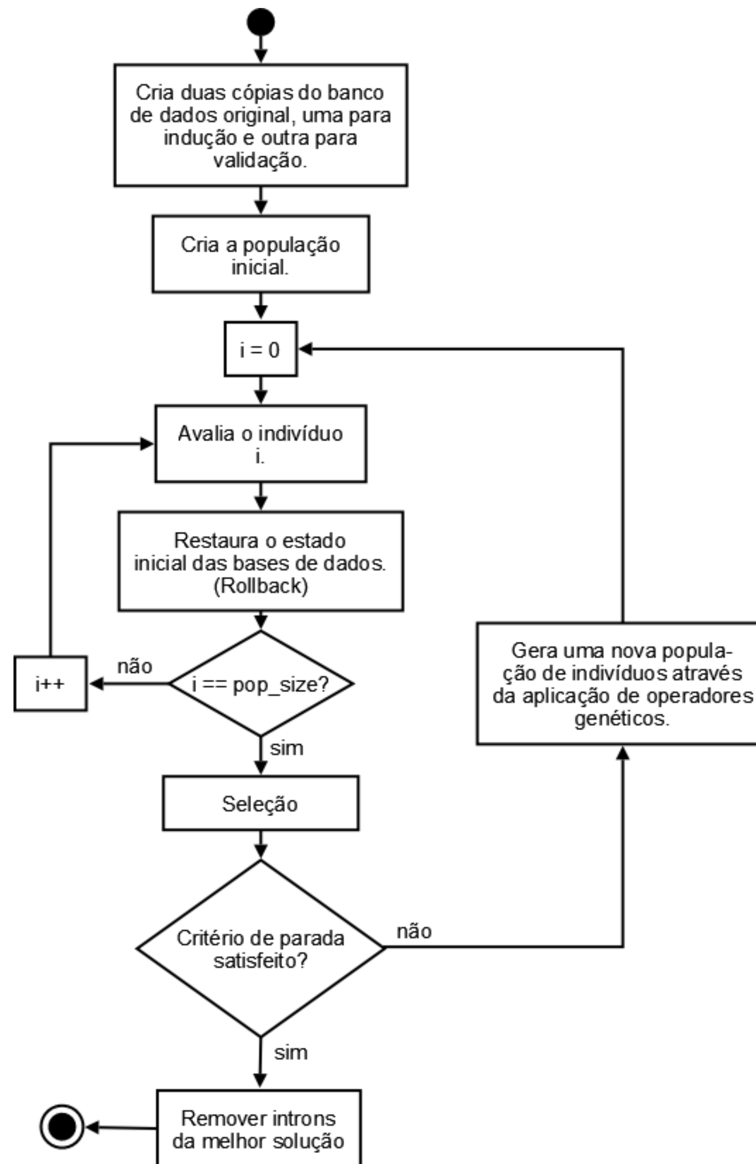

Fig. 5.3: Fluxograma do algoritmo de evolução de programas do LGPDB.

dados são revertidos à situação inicial para a avaliação do próximo indivíduo. Após a avaliação de todos os indivíduos, é realizada e seleção e a verificação se algum indivíduo soluciona o problema completamente. Se for o caso, o processo evolucionário é finalizado e os introns são removidos para se obter uma solução com maior grau de interpretabilidade. Caso nenhum candidato satisfaça o critério de parada, os operadores genéticos são aplicados, uma nova população é gerada e o processo é repetido até chegar na solução desejada ou o número de gerações superar um valor máximo, parâmetro previamente especificado.

5.2 Sistema de Gerenciamento de Biblioteca

Neste experimento, programas de computador são gerados automaticamente com o objetivo de prover um subconjunto de funcionalidades para manipulação de um banco de dados de um sistema simplificado de gerenciamento de biblioteca. Considerando uma arquitetura do tipo MVC (*Model-view-controller*) (Krasner and Pope, 1988) em que a aplicação é dividida nas camadas modelo, controlador e visão, responsáveis por modelar o estado atual da aplicação, implementar lógica de negócio ou controle, e apresentar uma interface de interação com o usuário; os programas induzidos pelo LGPDB implementam as duas primeiras camadas. O desenvolvimento de interfaces contendo componentes gráficos para especificação dos valores de entrada dos programas induzidos neste experimento finalizaria a implementação da arquitetura MVC. Porém, a geração automática de programas pra prover interfaces gráficas não é abordado pelo LGPDB.

Este experimento foi dividido em cinco etapas: modelagem do sistema, definição do banco de dados, escolha das funcionalidades que serão geradas automaticamente, elaboração dos casos de avaliação e finalmente a indução dos programas de computador.

5.2.1 Modelagem do Sistema

Conforme mostrado no Capítulo 2, independentemente da metodologia de desenvolvimento de software adotada, todo projeto começa com o levantamento de requisitos do sistema. Nesta etapa são levantadas quais funcionalidades serão suportadas pelo sistema. Uma vez que o objetivo deste experimento é validar conceitos de geração automática de programas para manipulação de informação que ainda precisam ser amadurecidos, esta etapa não considera todos os elementos de uma aplicação de mundo real. O enfoque desta etapa serão elementos básicos como, por exemplo, quais são as entidades do sistema e quais tipos de funcionalidade serão suportadas.

Um sistema para gerenciamento de biblioteca foi escolhido com propósito didático, uma vez que é intuitivo imaginar quais os tipos de informação e funcionalidades que o sistema irá prover. A Figura 5.4 mostra o diagrama de entidade-relacionamento (DER) (Chen, 1976) do sistema que será implementado. Como pode ser observado na modelagem, o sistema possui oito entidades: autor, livro, periódico, artigo, editora, usuário, tag e mensagem. No diagrama também são mostrados os atributos de cada entidade. No caso da entidade livro, há cinco atributos: id, título, ano, isbn e páginas. Além disso, este diagrama também mostra o relacionamento entre as entidades. A entidade usuário, por exemplo, pode fazer um empréstimo de um periódico que por sua vez contém artigos.

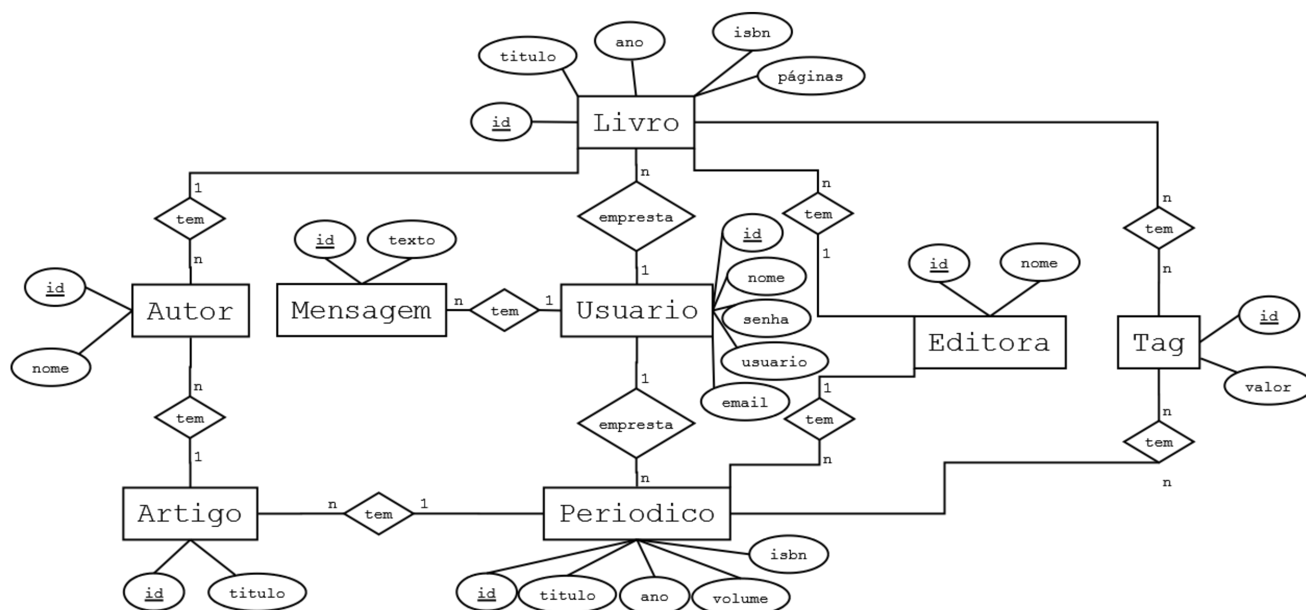

Fig. 5.4: Diagrama entidade-relacionamento do sistema simplificado de gestão de biblioteca.

5.2.2 Definição do Banco de Dados

Na subseção anterior foi realizada uma modelagem superficial do sistema de gerenciamento de bibliotecas. Nesta seção é definido como os dados serão representados num banco de dados relacional, seguindo o diagrama DER mostrado da Figura 5.4, conforme mostrado na Tabela 5.3. Para cada entidade do sistema, uma tabela foi criada no banco de dados, contendo seus atributos e chaves primárias e estrangeiras, mostradas em itálico, para permitir o relacionamento com outras entidades. Relacionamentos em que n registros de uma entidade podem se relacionar com n de outra, por exemplo, autor e livro, foram modelados utilizando uma tabela a parte, neste caso **autorLivroRel**, para realizar o mapeamento.

Utilizando a tabela anterior como referência, uma instância do banco de dados específico do LGPDB foi criada e todas as tabelas, com os seus respectivos atributos, foram adicionadas. Tendo o banco dados modelado e configurado, registros reais, obtidos na Internet, e registros hipotéticos, com relacionamentos necessários para modelar as funcionalidades do sistema, foram inseridos no banco de dados. Além disso, registros com relacionamentos inconsistentes como, por exemplo, um livro sem autor ou um periódico sem artigos, foram inseridos na base de dados para estimular os programas a explorarem as associações entre registros em tabelas distintas. Com o objetivo de maximizar o *fitness* removendo este tipo de registro indesejado na maioria dos casos de avaliação, programas exploram as associações entre tabelas e consequentemente ganham acesso a um maior número de atributos que podem ser utilizados para discriminar diferentes classes de registros, acelerando a convergência para solução ótima. Esta questão é abordada em maior profundidade na Seção 5.3.

Tab. 5.3: Tabelas e atributos do banco de dados relacional utilizado no experimento.

Tabela	Campos
usuario	*id*, nome, email, usuario, senha
autor	*id*, nome
livro	*id*, *editora_id*, titulo, paginas, ano, isbn
artigo	*id*, titulo, *periodico_id*
periodico	*id*, *editora_id*, ano, volume, numero, isbn
editora	*id*, nome
tag	*id*, valor
mensagem	*id*, texto
emprestimoLivro	*id*, *livro_id*, *usuario_id*
emprestimoPeriodico	*id*, *periodico_id*, *usuario_id*
autorLivroRel	*id*, *autor_id*, *livro_id*
autorArtigoRel	*id*, *autor_id*, *artigo_id*
tagRel	*id*, *livro_id*, *artigo_id*, *tag_id*
mensagemRel	*id*, *mensagem_id*, *usuario_id*

5.2.3 Escolha das funcionalidades

Dentre as diversas funcionalidades relacionadas a manipulação de informação, foi escolhido um subconjunto para mostrar a aplicabilidade do LGPDB. A Tabela 5.4, lista as funcionalidades que serão providas por programas gerados automaticamente para pesquisa, cadastro, inserção e atualização de registros. No caso de inserção e atualização, serão manipulados os relacionamentos ente os registros, portanto, os dados que serão relacionados já estão no banco de dados. A geração automática de programas para prover esse subconjunto de funcionalidades indica a viabilidade de geração de programas para funcionalidades semelhantes. Afinal, programas para adicionar uma *tag* à um livro ou a um periódico são induzidos num cenário muito semelhante.

5.2.4 Elaboração dos Casos de Avaliação

Tendo o banco de dados modelado e o conjunto de funcionalidades que serão providas por programas gerados automaticamente, a próxima etapa é definir o conjunto de casos de avaliação para cada uma das funcionalidades. Este conjunto deve apresentar exemplos de variáveis de entrada e o resultado esperado, este último podendo ser uma consulta ou o estado final do banco de dados da aplicação. A seguir são mostrados alguns exemplos de entrada e resultado desejado para cada tipo de manipulação de dados. Na prática, para cada funcionalidade que será abordada, é utilizado um conjunto com múltiplos exemplos deste tipo.

Na Tabela 5.5 é mostrado um exemplo de caso de avaliação para uma consulta em que é especifi-

Tab. 5.4: Funcionalidades do sistema simplificado para gestão de biblioteca que serão providas por programas gerados automaticamente.

ID	**Descrição**
Q1	Listar livros escritos pelo autor de nome X.
Q2	Listar usuários com empréstimos livros escritos pelo autor de nome X.
Q3	Listar usuários com empréstimos de periódicos contendo artigos escritos pelo autor de nome X.
Q4	Listar livros disponíveis para empréstimo, contendo a *tag* X.
D1	Remover livros publicados antes do ano X.
D2	Remover empréstimo do livro intitulado X realizado pelo usuário Y.
D3	Remover livros e artigos escritos pelo autor de nome X.
I1	Enviar a mensagem Y a todos os usuários com empréstimos do livro de título X.
I2	Adicionar a *tag* Y ao livro intitulado X.
I3	Criar o empréstimo do livro X pelo usuário Y.
U1	Atualizar a editora para X de livros com a editora Y.

cado um dado como entrada, nome de uma *tag*, e um conjunto de livros como resultado. Na Tabela 5.6 é mostrado um exemplo para abordar uma funcionalidade de exclusão. Neste caso, o resultado desejado é a exclusão de um conjunto de registros especificado utilizando o comando **delete(**$entidade$**,** $chave_primria$**)**, dado um nome de autor como entrada. Na Tabela 5.7 é mostrado um exemplo para o caso de inserção em que a entrada é o título de um livro e uma mensagem pré-definida no sistema que deve ser enviada a todos os usuários com empréstimo deste livro. Portanto, neste último caso, o resultado desejado para cada valor de entrada é a inserção de um novo registro, que associa um usuário com uma mensagem, utilizando o comando **insert(**$entidade$**,** $\{atributos\}$. Finalmente, na Tabela 5.8 é mostrado um exemplo de atualização tendo como entrada a associação atual e a nova associação desejada e, como resultado, a atualização dos registros especificados através do comando **update(**$entidade$**,** $chave_primaria$**,** $campo$**,** $valor$**)**. Os comandos **insert**, **delete** e **update** utilizados para especificar os resultados desejados são providos pelo SGBD utilizado pelo LGPDB.

Apesar do resultado desejado, no caso de funcionalidades que alteram os dados do banco de dados, ser representado como um conjunto de comandos que manipulam um banco de dados, não há semelhança entre esta representação e a solução desejada. Diferentemente dos comandos utilizados nos casos de avaliação, os programas induzidos pelo LGPDB devem encontrar uma associação entre a entrada e saída, para modelar a funcionalidade de forma correta, considerando diferentes exemplos de entrada e resultados desejados.

Tab. 5.5: Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **Q4**.

Funcionalidade	Q4
Descrição	Listar livros disponíveis para empréstimo, contendo a *tag* X.
Entrada	“inteligência artificial”
Tipo	Consulta
Resultado	{“0”, “Inteligência Artificial: uma abordagem moderna”, “0”, “1132”, “2009”, “0137903952”}, {“1”, “Paradigmas de prog. de inteligência artificial”, “1”, “946”, “1991”, “1558601910”}, {“2”, “Aprendizado de Máquina”, “2”, “432”, “1997”, “0070428077”}, {“3”, “Programação Genética”, “3”, “840”, “1992”, “0262111705”}

Tab. 5.6: Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **D3**.

Funcionalidade	D3
Descrição	Remover livros e artigos escritos pelo autor de nome X.
Entrada	“Peter Norvig”
Tipo	Exclusão
Resultado	delete(livro, 0) delete(livro, 1) delete(artigo, 142)

Tab. 5.7: Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **I1**.

Funcionalidade	I1
Descrição	Enviar a mensagem Y a todos os usuários com empréstimos do livro de título X.
Entrada	“Aprendizado de Máquina”, “Novo livro disponível sobre assunto semelhante: Inteligência Artificial.”
Tipo	Inserção
Resultado	insert(mensagemRel, 5, 7) insert(mensagemRel, 5, 8) insert(mensagemRel, 5, 9)

5.2.5 Indução dos Programas de Computador

As quatro etapas anteriores prepararam o ambiente para indução dos programas de computador. Finalmente, nesta última etapa, são definidos os parâmetros do processo de indução e programas são gerados para modelar cada funcionalidade. Para estipular os parâmetros número de **ResultSet** disponíveis nrs, tamanho da população tam_{pop}, tamanho máximo de programa $tamprog_{max}$, probabilidade de cruzamento $pcruz$ e probabilidade de mutação $pmut$, foram realizados testes empíricos utilizando

Tab. 5.8: Exemplo de caso de avaliação utilizado na indução de programas para prover a funcionalidade **U1**.

Funcionalidade	U1
Descrição	Atualizar a editora para X de livros com a editora Y.
Entrada	“Springer”, “Springer-Verlag”
Tipo	Inserção
Resultado	update(livro, 13, editora_id, 10) update(livro, 14, editora_id, 10) update(livro, 15, editora_id, 10)

as funcionalidades **Q1**, **D1**, **I1** e **U1**, visando a configuração com a convergência mais rápida para a solução ótima. Por meio destes experimentos obteve-se nrs = 4, tam_{pop} = 1000, $tamprog_{max}$ = 20, $pcruz$ = 0.3 e $pmut$ = 0.9.

Com todo o ambiente definido é possível analisar a complexidade do problema, por exemplo, analisando o tamanho do espaço combinatório de soluções. Neste problema em questão, a instrução **select(Table tb, ResultSet rs)** pode ser parametrizada de 56 formas distintas, uma vez que existem 14 tabelas e 4 objetos de **ResultSet** em memória. Por sua vez, a instrução **Related(ResultSet rs1, ResultSet rs2)** pode ser parametrizada de 12 formas distintas. No caso da instrução **Filter(ResultSet rs, Attribute attr, Rule r, InputValue v)**, o número de atributos **attr** depende da tabela e o número de valores de entrada depende da funcionalidade a ser modelada. No caso da filtragem de um **ResultSet** contendo registros da tabela **livro** e apenas um valor de entrada, esta função pode ser parametrizada de 64 formas distintas, uma que vez que há 4 **ResultSet** em memória, 4 atributos para a tabela **livro** e 4 regras de filtragem. Considerando todas as tabelas do banco de dados que têm atributos além de chaves, a instrução **Filter** pode ser parametrizada de 256 formas distintas. Realizando esta análise para todas as instruções, uma única posição de programa pode ser configurada de 588 formas distintas, considerando 14 tabelas e seus atributos, e uma variável de entrada. Portanto, neste caso, e considerando a indução de programas com tamanho máximo de 20 instruções, o espaço combinatório de soluções é equivalente à 588^{20} ou aproximadamente 10^{55}.

Utilizando os parâmetros determinados empiricamente, programas para proverem as funcionalidades apresentadas na Tabela 5.4 foram induzidos. A Tabela 5.9 mostra, para cada funcionalidade, um dos programas obtidos dentre as 20 execuções realizadas, média (Gen_{avg}), desvio padrão (Gen_{std}), mediana (Gen_{median}) do número de gerações necessárias para chegar a solução ótima, e o número de tentativas que não convergiram para a solução ótima.

Os resultados indicam a viabilidade do uso de programação genética para modelagem de processos que manipulam registros armazenados num banco de dados relacional. A indução de programas para proverem as funcionalidades **D2**, **D3** e **I1** não apresentaram convergência para solução ótima em todas as execuções. Entretanto, considerando aplicações práticas, no caso de uma tentativa não

Tab. 5.9: Resultado da indução de programas para prover um subconjunto de funcionalidades de um sistema de gerenciamento de biblioteca.

ID	Programa	Gen_{avg}	Gen_{std}	Gen_{median}	NC
Q1	select(autor,rs3) select(autorLivroRel,rs2) filter(nome,equals,x,rs3) related(rs2,rs3) select(livro,rs1) related(rs1,rs2)	75	34	79	0
Q2	select(autor,rs2) filter(nome,equals,x,rs2) select(autorLivroRel,rs1) related(rs1,rs2) select(livro,rs3) select(emprestimoLivro,rs2) related(rs3,rs1) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)	283	161	265	0
Q3	select(autor,rs3) select(artigo,rs1) filter(nome,equals,x,rs3) select(autorArtigoRel,rs2) related(rs2,rs3) related(rs1,rs2) select(emprestimoPeriodico,rs2) select(periodical,rs3) related(rs3,rs1) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)	430	205	365	0
Q4	select(tag,rs4) filter(valor,equals,x,rs4) select(livro,rs1) select(tagRel,rs2) related(rs2,rs4) related(rs1,rs2) select(emprestimoLivro,rs3) unrelated(rs1,rs3)	216	134	198	0
D1	select(livro,rs2) filter(ano,less,x,rs2) delete(livro,rs2)	28	31	14	0
D2	select(emprestimoLivro,rs2) select(livro,rs3) filter(titulo,equals,x,rs3) related(rs2,rs3) select(usuario,rs3) filter(nome,equals,y,rs3) related(rs2,rs3) delete(emprestimoLivro,rs2)	715	495	746	6
D3	select(artigo,rs3) select(autor,rs4) filter(nome,equals,x,rs4) select(autorLivroRel,rs1) select(livro,rs2) relate(rs1,rs4) relate(rs2,rs1) select(autorPapelRel,rs4) select(autor,rs1) delete(livro,rs2) filter(nome,equals,x,rs1) relate(rs4,rs1) relate(rs3,rs4) delete(artigo,rs3)	402	374	279	3
I1	select(livro,rs2) filter(title,equals,x,rs2) select(emprestimoLivro,rs3) related(rs3,rs2) select(usuario,rs2) related(rs2,rs3) select(mensagem,rs1) createRelation(mensagemRel,rs3,rs2) filter(texto,equals,y,rs1) setRelation(rs3,rs1)	709	391	662	1
I2	select(tag,rs2) filter(valor,equals,y,rs2) createRelation(tagRel,rs3,rs2) select(livro,rs4) filter(nome,equals,x,rs4) setRelation(rs3,rs4)	80	49	69	0
I3	select(usuario,rs1) filter(nome,equals,x,rs1) createRelation(emprestimoLivro,rs3,rs1) select(livro,rs1) filter(nome,equals,y,rs1) setRelation(rs3,rs1)	120	51	119	0
U1	select(editora,rs2) filter(nome,equals,x,rs2) select(livro,rs4) select(editora,rs3) filter(nome,equals,y,rs3) related(rs4,rs3) setRelation(rs4,rs2)	209	201	151	0

convergir, basta repetir o processo de indução até se obter a solução ótima. A modelagem da funcionalidade **D3**, em que registros são excluídos das tabelas **livro** e **artigo**, mostra que LGPDB é capaz de induzir programas que alteram múltiplas tabelas. O experimento também comprova que o método de remoção de introns, que ocorre ao final do processo evolucionário, permite a geração de programas coesos e com boa interpretabilidade. Neste experimento foi abordada apenas uma funcionalidade de atualização, dada a limitação de operar apenas nos relacionamentos entre registros. Na seção seguinte será analisada uma abordagem que combina múltiplas formas de manipulação de dados num único programa, em que a atualização de registro irá desempenhar um papel fundamental.

Em relação ao tempo necessário para a indução de programas, a técnica se mostra viável uma vez que, dentre todas tentativas, o processo evolucionário mais rápido, referente a funcionalidade **D1**, demorou 9 segundos, e o processo mais lento, referente a funcionalidade **U1**, demorou 19 minutos. A maior demora na indução de funcionalidades de atualização, assim como de inserção, está relacionada ao fato de que, neste caso, a execução de cada geração do processo evolucionário é mais lenta. Isto ocorre pois nestes casos é necessário comparar e manipular atributos individuais dos registros, tornando as operações de avaliação e recuperação de estados do banco de dados mais lentas. Este experimento foi realizado utilizando um computador pessoal equipado com um processador de dois núcleos *AMD Turion 64 X2* com frequencia de 1.8 GHz e com 2GB de memória principal.

Estágios de evolução de programas para manipular os dados de formas distintas, no intuito de prover as funcionalidades **Q4**, **D3**, **I1**, **U1**, são mostrados nas Figuras 5.5, 5.6, 5.7, 5.8. Nestes exemplos, com o intuito de aumentar a interpretabilidade, os introns foram removidos, permanecendo apenas as instruções efetivas. Como pode ser observado nestes exemplos, inicialmente, os programas operam um conjunto de dados incluindo todos os registros desejados e diversos registros indesejáveis. Em seguida, instruções para relacionar registros em diferentes tabelas e para filtrar atributos são utilizadas para remover os registros não desejados do conjunto de registros manipulados. No caso da indução de um programa para prover a funcionalidade **Q4**, mostrado na Figura 5.5, até a geração 192, registros indesejáveis foram removidos por meio de relacionamento de registros de diferentes tabelas. Neste caso, antes mesmo de filtrar um livro pela tag, registros de livros com empréstimo ou sem tag, já podem ser desconsiderados. Finalmente, na geração 193, após o programa relacionar todos os registros corretamente, é realizado um filtro utilizando o valor da tag.

No caso da indução de um programa para exclusão de registros, como mostrado na Figura 5.6, as soluções das primeiras gerações praticamente excluem todos os registros das tabelas alvo (livro, periodico). Enquanto o atributo discriminante não está acessível, neste caso o nome do autor, o programa explora os relacionamentos para remover os registros que não satisfazem as associações corretas. Finalmente, na geração 371, é realizada uma filtragem e apenas os registros corretos são excluídos.

A operação de inserção de registros não pode ser feita utilizando uma única instrução. Portanto, a instrução **createRelation()** é utilizada para criar os registros e a instrução **setRelation()** é utilizada para atualizar os relacionamentos de registros. Inicialmente, múltiplos registros são criados, incluindo um subconjunto semelhante aos registros desejados. Após a realização dos relacionamentos e filtragens corretas, apenas os registros desejados são criados e associados com os registros relacionados com o parâmetro da funcionalidade.

No caso da atualização, mostrado na Figura 5.8, o primeiro programa que afeta o *fitness* associa todos os livros com todas as editoras. Neste caso, a maximização do *fitness* ocorre pois no cálculo da distância é considerado se os registros alvo foram alterados de alguma forma. Seguindo a mesma lógica dos exemplos anteriores, filtragens são realizadas ao longo das gerações até que apenas os registros corretos sejam atualizados.

A Figura 5.9 mostra o valor da função de adaptação ao longo das gerações, relativa aos processos evolucionários para a geração de programas para prover as funcionalidades **Q4**, **D3**, **I1**, **U1**. Independentemente da tarefa, nas primeiras gerações, programas identificam a tabela alvo e realizam uma primeira operação que atua nos registros desejados, seja uma consulta ou uma alteração. No caso da indução de um programa para prover a funcionalidade **Q4**, o melhor programa da geração 7 consulta todos os livros associados a uma *tag*. Portanto, o conjunto resultado apresenta todos os registros desejados misturados aos falsos positivos. Visto que na avaliação de programas para consulta os falsos negativos recebem uma maior penalização, aquele programa consegue aumentar significativamente o valor de *fitness* (aproximadamente para 0.8), conforme pode ser observado no gráfico. Este fenômeno ocorreu na indução de todas as funcionalidades abordadas neste experimento. Após esta primeira etapa, operando num conjunto de menor cardinalidade, programas candidatos filtram e associam registros para finalmente modelar a operação desejada. As gerações em que ocorreu um aumento do valor da função de adaptação são destacadas em vermelho. Intervalos maiores sem aumento do valor desta função surgem após os programas atingirem um número significativo de instruções efetivas. Quanto maior o número de instruções efetivas, maior a probabilidade de um operador genético afetar negativamente um trecho do programa relevante para a solução. Isto pode ocorrer tanto manipulando desnecessariamente instruções efetivas, como alterando variáveis em memória utilizadas pelas mesmas.

Apesar da simplicidade, funcionalidades semelhantes as apresentadas neste experimento estão presentes em muitos sistemas e continuam sendo implementadas manualmente. Num cenário prático, as funcionalidades poderiam ser modeladas por um analista de sistemas, utilizando exemplos de operações, e providas por programas gerados de forma automática. Em ambientes de desenvolvimento que utilizam metodologias em que os casos de teste são desenvolvidos antes das funcionalidades, por exemplo *Test-driven Development*, LGPDB poderia automatizar a implementação de funcionalidades

geração: 1
select(livro,rs1)

geração: 7
select(livro,rs1) select(tagRel,rs2) relate(rs1,rs2)

geração: 13
select(livro,rs1) select(tagRel,rs2) relate(rs1,rs2) select(autorLivroRel,rs4) relate(rs1,rs4)

geração: 19
select(livro,rs1) select(autorLivroRel,rs4) relate(rs1,rs4) select(emprestimoLivro,rs2) not_relate(rs1,rs2)

geração: 23
select(livro,rs1) select(tagRel,rs4) relate(rs1,rs4) select(autorLivroRel,rs4) relate(rs1,rs4) select(emprestimoLivro,rs2) not_relate(rs1,rs2)

geração: 149
select(tag,rs3) select(tagRel,rs4) select(livro,rs1) select(emprestimoLivro,rs2) not_relate(rs1,rs2) relate(rs4,rs3) relate(rs1,rs4) select(autorLivroRel,rs4) relate(rs1,rs4)

geração: 156
select(tag,rs3) select(tagRel,rs4) filter(2005,equals,x,rs3) select(livro,rs1) select(emprestimoLivro,rs2) not_relate(rs1,rs2) relate(rs4,rs3) relate(rs1,rs4)

Fig. 5.5: Estágios de evolução de um programa para prover a funcionalidade de consulta **Q4**.

geração: 6
select(livro,rs2) delete(livro,rs2)

geração: 26
select(livro,rs2) delete(livro,rs2) select(artigo,rs3) delete(artigo,rs3)

geração: 72
select(livro,rs2) select(editora,rs4) relate(rs2,rs4) delete(livro,rs2) select(artigo,rs3) delete(artigo,rs3)

geração: 85
select(livro,rs2) select(emprestimoLivro,rs4) select(artigo,rs3) select(tagRel,rs1) relate(rs2,rs1) relate(rs2,rs4) delete(livro,rs2) delete(artigo,rs3)

geração: 185
select(livro,rs2) select(artigo,rs3) select(autor,rs4) select(autorLivroRel,rs1) filter(nome,equals,x,rs4) relate(rs1,rs4) relate(rs2,rs1) select(autorArtigoRel,rs4) delete(livro,rs2) relate(rs3,rs4) delete(artigo,rs3)

geração: 356
select(artigo,rs3) select(autor,rs4) filter(nome,equals,x,rs4) select(autorLivroRel,rs1) select(livro,rs2) relate(rs1,rs4) relate(rs2,rs1) select(autorArtigoRel,rs4) select(autor,rs1) delete(livro,rs2) relate(rs4,rs1) relate(rs3,rs4) delete(artigo,rs3)

geração: 371
select(artigo,rs3) select(autor,rs4) filter(nome,equals,x,rs4) select(autorLivroRel,rs1) select(livro,rs2) relate(rs1,rs4) relate(rs2,rs1) select(autorArtigoRel,rs4) select(autor,rs1) delete(livro,rs2) filter(nome,equals,x,rs1) relate(rs4,rs1) relate(rs3,rs4) delete(artigo,rs3)

Fig. 5.6: Estágios de evolução de um programa para prover a funcionalidade de exclusão **D3**.

geração: 1
select(usuario,rs4) createRelation(mensagemRel,rs1,rs4)

geração: 21
select(mensagem,rs3) select(usuario,rs4) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

geração: 27
select(mensagem,rs3) filter(texto,equals,y,rs3) select(usuario,rs4) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

geração: 93
select(mensagem,rs3) select(usuario,rs4) select(emprestimoLivro,rs1) filter(texto,equals,y,rs3) relate(rs4,rs1) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

geração: 319
select(mensagem,rs3) select(usuario,rs4) select(emprestimoLivro,rs1) filter(texto,equals,y,rs3) relate(rs4,rs1) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

geração: 377
select(mensagem,rs3) select(usuario,rs4) select(emprestimoLivro,rs1) filter(texto,equals,y,rs3) select(livro,rs2)relate(rs1,rs2)relate(rs4,rs1) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

geração: 378
select(mensagem,rs3) select(usuario,rs4) select(emprestimoLivro,rs1) filter(texto,equals,y,rs3) select(livro,rs2)filter(titulo,equals,x,rs2) relate(rs1,rs2)relate(rs4,rs1) createRelation(mensagemRel,rs1,rs4) setRelation(rs1,rs3)

Fig. 5.7: Estágios de evolução de um programa para prover a funcionalidade de inserção **I1**.

geração: 39
select(livro,rs1) select(editora,rs2) setRelation(rs2,rs1)

geração: 75
select(livro,rs2) select(editora,rs1) relate(rs1,rs2) select(livro,rs2) setRelation(rs2,rs1)

geração: 83
select(editora,rs2) select(livro,rs1) relate(rs1,rs2) select(livro,rs2) relate(rs2,rs1) setRelation(rs2,rs1)

geração: 86
select(editora,rs2) select(livro,rs1) filter(nome,equals,y,rs2) relate(rs1,rs2) select(livro,rs2) setRelation(rs2,rs1)

geração: 228
select(editora,rs1) filter(nome,equals,y,rs1) select(livro,rs2) relate(rs2,rs1) select(editora,rs1) filter(nome,equals,x,rs1) setRelation(rs2,rs1)

Fig. 5.8: Estágios de evolução de um programa para prover a funcionalidade de atualização **U1**.

de menor complexidade, utilizando os casos de teste para avaliação de soluções candidatas. Esta estratégia permitiria que programadores humanos se dedicassem às tarefas mais complexas.

5.3 Propriedades de Banco de Dados de Indução

Uma premissa para que o processo de evolução de programas utilizando o LGPDB seja bem sucedido é a utilização de um banco de dados com propriedades estatísticas suficientes para, pelo aprendizado por indução, encontrar os relacionamentos entre entrada e saída desejados. Isto significa que o banco de dados deve apresentar uma quantidade e uma variedade de registros capazes de exemplificar as diversas associações que serão exploradas por programas gerados automaticamente. Quanto maiores o número e a variação dos registros no banco de dados, menor é a probabilidade de se modelar uma relação entre entrada e saída incorreta, na perspectiva da funcionalidade que está sendo modelada por programas gerados automaticamente. Em outras palavras, menor será a chance de se encontrar uma segunda forma de modelar a relação entre entrada e saída, que não apresente a funcionalidade desejada, mas que obtenha o valor máximo no processo de avaliação.

No caso do experimento apresentado na Seção 5.2, a utilização de um banco de dados sem tais propriedades poderia levar à indução incorreta de programas. Supondo um cenário em que se deseja modelar a funcionalidade **F1** para listar os livros publicados pela editora X, escritos pelo autor Y, e que, por limitação do banco de dados, todos os autores estão associados apenas a uma editora, o processo de indução de programas poderá modelar a funcionalidade incorretamente. Uma solução

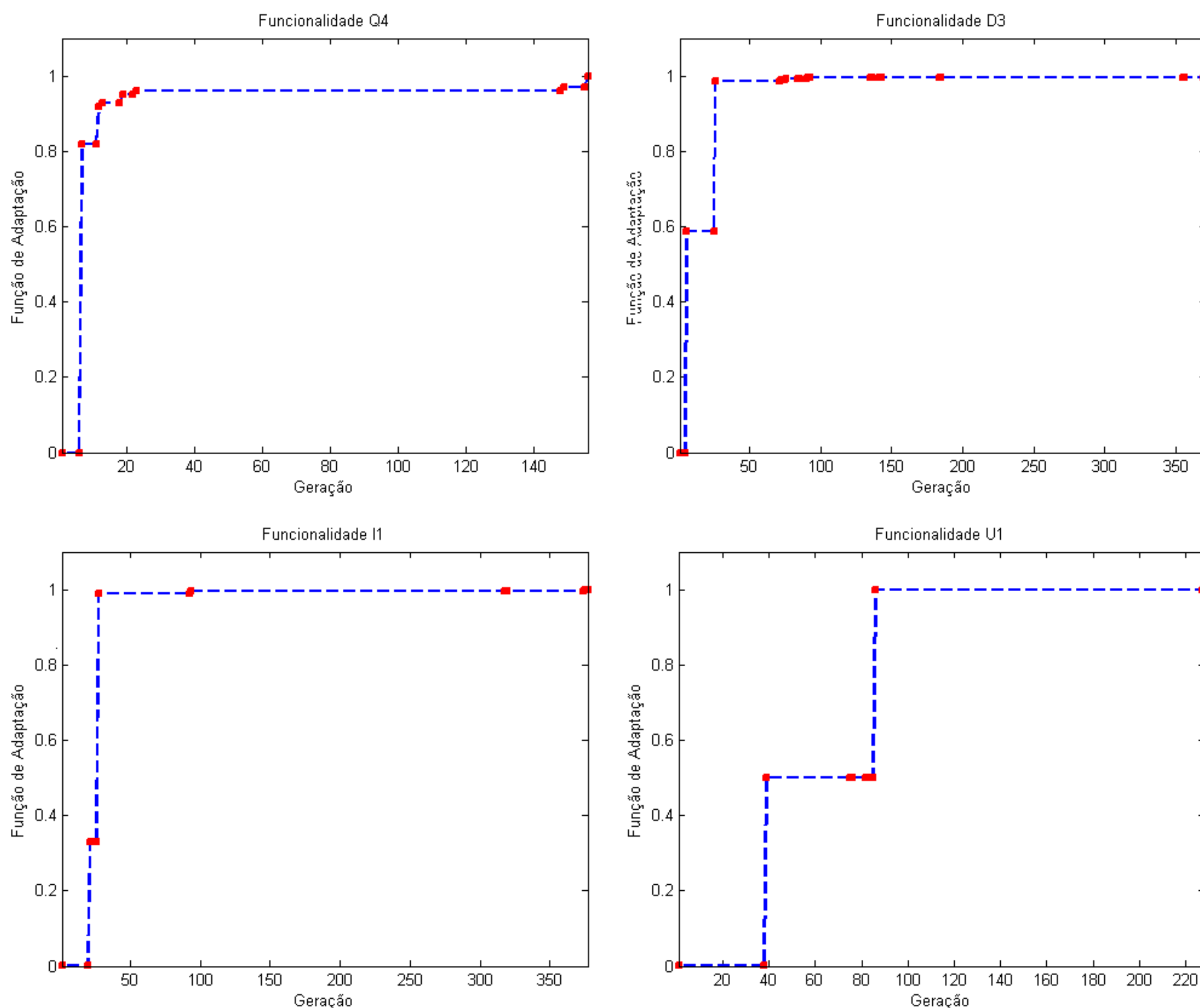

Fig. 5.9: Gráfico dos valores da função de adaptação por geração, relativos às funcionalidades **Q4**, **D3**, **I1**, **U1**. Os pontos em vermelho indicam as gerações em que ocorreu variação do valor da função de adaptação.

que lista os livros utilizando apenas o nome do autor terá a nota máxima no processo de avaliação uma vez que, dada a limitação do banco de dados, é capaz de modelar corretamente a relação entre entrada e saída sem utilizar a variável de entrada "editora". Quando o programa resultante for utilizado numa aplicação real, em que o banco de dados não apresenta esta limitação - todos os autores estarem associados apenas a uma editora - ele irá apresentar mau funcionamento.

Nessa questão, se comparado com bancos de dados utilizados por abordagens de automatização de testes unitários, bancos de dados utilizados para indução de programas com LGPDB exigem um maior cuidado em relação às propriedades estatísticas dos dados. Apesar deste problema também es-

tar presente nos métodos de automatização de testes, permitindo que programas desenvolvidos por um programador humano sejam aprovados no processo de teste mesmo que explorem relacionamentos indesejáveis, o impacto neste caso é menor. Isto ocorre devido à abordagem humana e à abordagem por aprendizado indutivo atacarem o problemas de formas totalmente distintas. O programador humano, provido de alta capacidade cognitiva, consegue de forma *top-down* entender conceitos de alto nível que definem o problema e formular uma solução, exigindo apenas a avaliação de algumas soluções candidatas por ferramentas de teste automatizadas. Por outro lado, a abordagem por aprendizado indutivo, que é desprovida de ou incorpora pouca capacidade cognitiva, atua de forma *bottom-up*, partindo das propriedades dos dados, podendo realizar milhões de avaliações de soluções para tenta encontrar uma forma de modelar o relacionamento de entrada e saída dos exemplos utilizados no processo de indução. Sendo assim, considerando o exemplo de obtenção da funcionalidade **F1** num banco de dados, em que todos os autores estão relacionados com apenas uma editora, e aplicando uma metodologia de automatização de teste, espera-se que o programador humano resolva o problema corretamente, pois para ele será lógica a necessidade de utilizar as variáveis de entrada "editora" e "autor" na solução do problema.

Outra premissa para indução de programas utilizando o LGPDB é a presença de registros descorrelacionados no banco de dados. Estes registros são necessários para discriminar a qualidade de diferentes soluções nas fases iniciais do processo de indução, em que os atributos que precisam ser relacionados às variáveis de entrada ainda não estão acessíveis aos programas, uma vez que para acessá-los é necessário modelar relacionamentos entre diferentes tabelas no banco de dados. Para listar usuários com empréstimo de livros, utilizando o banco de dados do experimento apresentado na Seção 5.2, é necessário relacionar as tabelas *usuario* e *emprestimoLivro* para remover usuários sem empréstimos. Este tipo de desassociação entre duas entidades é aceitável, dada a modelagem do sistema. Entretanto, outros tipos de desassociação entre registros como, por exemplo, um livro sem autor ou um periódico sem artigos, não fazem sentido, dada a modelagem do sistema. A inexistência deste último tipo de desassociação impede que soluções candidatas sejam recompensadas por explorar os relacionamentos entre algumas tabelas do banco de dados. No caso dos exemplos anteriores, relacionar as tabelas de livros a autores ou periódicos a artigos não filtraria nenhum registro, dado que todo livro teria autor e todo periódico teria artigo. Portanto, registros com este tipo de desassociação são inseridos na base de dados para aumentar a capacidade de discriminação de soluções pelas funções de avaliação.

A Figura 5.10 mostra um grafo com as associações entre registros de diferentes tabelas no banco de dados utilizado no experimento da Seção 5.2. Em verde, são mostrados os registros reais, obtidos de bases de livros e artigos disponíveis na Internet. Em vermelho, são mostrados registros hipotéticos e com algumas desassociações. Cada agrupamento de nós do grafo representa um conjunto de

registros de uma determinada tabela do banco de dados. Cada nó tem uma sigla que indica o nome da tabela, na forma: usuario (**US**), autor (**AU**), livro (**BO**), artigo (**PA**), periodico (**PE**), editora (**PU**), emprestimoLivro (**BL**), emprestimoPeriodico **(PL)**, autorLivroRel (**AB**), autorArtigoRel (**AP**), tag (**TA**), tagRel (**TR**), mensagem (**ME**), mensagemRel (**MR**). As arestas entre nós do grafo indicam os relacionamentos entre registros, criados por meio de chaves. O objetivo deste grafo é mostrar que os dados com desassociações não estão relacionados com os dados reais do banco de dados. Portanto, estes são necessários apenas no processo de indução. Na aplicação real, os programas irão funcionar mesmo sem a presença deste conjunto de dados.

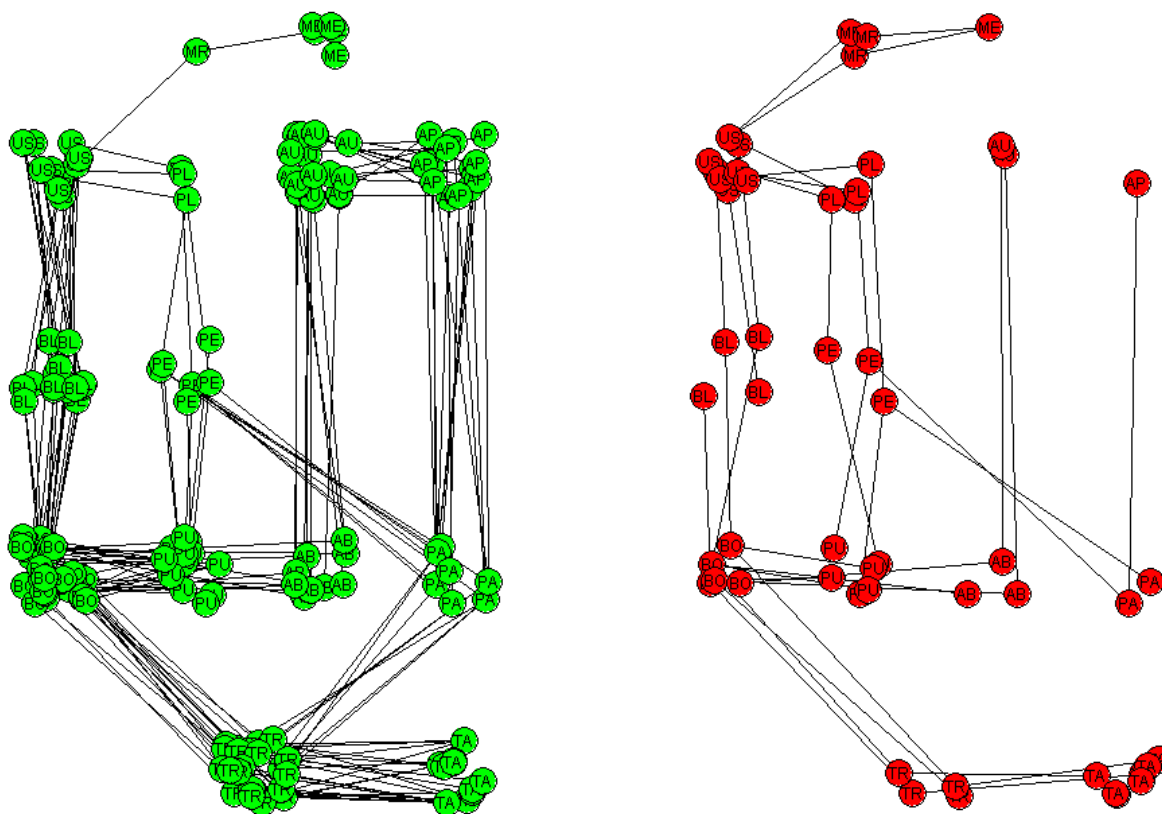

Fig. 5.10: Grafo das associações de registros de tabelas no banco de dados. Os nós são registros de uma dada tabela e arestas representam relacionamentos, criados por meio de chaves. Em verde, são mostrados os registros reais, obtidos de bases de livros e artigos disponíveis na Internet. Em vermelho, são mostrados registros hipotéticos e com algumas desassociações.

Para analisar o impacto destes dados na indução de um programa genético, a Figura 5.11 mostra a evolução de um programa para prover a funcionalidade **Q3**: “Listar usuários com empréstimos de periódicos contendo artigos escritos pelo autor de nome X”. Neste caso de indução de programa, a maior parte do processo evolucionário está relacionada a encontrar o conjunto de relacionamentos correto para associar um usuário com um empréstimo de livro. A partir da geração 364, em que

geração: 1, fitness padronizado: 270.0
// seleciona todos os usuários
select(usuario,rs1)

geração: 15, fitness padronizado: 90.0
// seleciona todos os usuários com empréstimo de periódico
select(emprestimoPeriodico,rs2) select(usuario,rs1) related(rs1,rs2)

geração: 62, fitness padronizado: 75.0
// seleciona todos os usuários com empréstimo de periódico - desconsidera empréstimo sem periódico
select(emprestimoPeriodico,rs2) select(periodico,rs3) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)

geração: 148, fitness padronizado: 60.0
// seleciona todos os usuários com empréstimo de periódico - desconsidera periódico sem artigo
select(paper,rs1) select(emprestimoPeriodico,rs2) select(periodico,rs3) related(rs3,rs1) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)

geração: 364, fitness padronizado: 45.0
// seleciona todos os usuários com empréstimo de periódico - desconsidera artigo sem autor
select(paper,rs1) select(autorArtigoRel,rs2) related(rs1,rs2) select(emprestimoPeriodico,rs2) select(periodico,rs3) related(rs3,rs1) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)

geração: 401, fitness padronizado: 0.0
// seleciona todos os usuários com empréstimo de periódico que tem artigo escrito pelo autor X
select(autor,rs3) select(paper,rs1) filter(nome,equals,x,rs3) select(autorArtigoRel,rs2) related(rs2,rs3) related(rs1,rs2) select(emprestimoPeriodico,rs2) select(periodico,rs3) related(rs3,rs1) related(rs2,rs3) select(usuario,rs1) related(rs1,rs2)

Fig. 5.11: O impacto causado no processo evolucionário por registros com desassociações.

os devidos relacionamentos já foram encontrados, programas candidatos conseguem acessar corretamente os atributos de autores relacionados a artigos de periódicos com empréstimos. Portanto, todo o processo evolucionário que precede essa geração tem seus programas discriminados pela capacidade de remover os registros com desassociações. Apenas a partir da geração 364 o atributo de entrada nome do autor pode ser utilizado de forma correta para maximizar o *fitness* de indivíduos.

5.4 Melhorias e Experimentos Preliminares

O experimento apresentado na seção anterior mostrou que é possível induzir programas capazes de modelar processos de manipulação de informação. Naquele experimento, o banco de dados é tratado como uma entidade dinâmica, portanto, o seu estado pode ser alterado por um dado processo que deve ser modelado. Além da validar o principal diferencial de LGPDB, o experimento anterior

também evidenciou limitações e pontos a melhorar. Esta seção apresenta experimentos preliminares em questões que deverão ser tratadas em maior profundidade em trabalhos futuros.

5.4.1 Combinação de Regras

Nos experimentos apresentados na Seção 5.2, o desafio estava em relacionar as variáveis de entrada com registros do sistema e operar estes registros de forma correta, dado um processo ilustrado em casos de avaliação. Entretanto, com o conjunto de instruções apresentado na Seção 5.1.2, registros são selecionados combinando múltiplas regras de filtragem, sendo que todas as regras precisam ser satisfeitas. Com o intuito de permitir a modelagem de regras com operador **OU**, foram criadas duas instruções, uma para adição de regras (**AddRule**) e outra para filtragem (**Filter_2**) que utiliza um objeto com múltiplas regras. Em relação a processos de atualização de dados do experimento anterior, programas eram capazes de atualizar os relacionamentos entre registros por meio da instrução **setRelation**. Porém, a alteração de atributos, por exemplo, utilizando valores passados como parâmetro, não era possível. Para este propósito foi criada a instrução **SetValue** que permite alterar o valor de um atributo de um **ResultSet** em função de um valor em memória. Estas novas instruções são listadas abaixo:

- **addRule(Operator op, Attribute attr, Rule r, InputValue v, RuleObject ro)**: adiciona a regra **r**, associada com o atributo **attr** e variável **v**, ao objeto de regras **ro**, utilizando operador **op**.
- **Filter_2(ResultSet rs, RuleObject ro):** filtra o conjunto de registros em **rs** considerando o conjunto de regras armazenado em **ro**. Esta instrução substitui a instrução **Filter()**.
- **setValue(Attribute attr, InputValue v, Operation o, ResultSet r)**: altera o valor do atributo **attr**, para os registros em **rs**, utilizando a operação **o** e o valor **v**. A operação **o** pode ser atribuição, incremento e decremento.

Para testar as novas instruções de filtragem, foi realizada a indução de uma nova funcionalidade: “Listar usuários com empréstimos de livros escritos pelos autores **X** ou **Y**”. Foi utilizada a mesma metodologia do experimento anterior. O programa resultante é apresentado abaixo:

5.4.2 Composição de soluções utilizando múltiplos programas

No caso dos experimentos apresentados em 5.2, programas são capazes de modelar processos que consultam, inserem, excluem ou atualizam registros no banco de dados. Porém, em muitos casos, um mesmo processo pode manipular os dados de múltiplas formas, por exemplo, realizando inserções e

```
01:  select(autor,rs3)
02:  addRule(_,nome,equals,X,rule1)
03:  select(autorLivroRel,rs2)
04:  addRule(or,nome,equals,Y,rule1)
05:  filter_2(rs3,rule1)
06:  select(usuario,rs1)
07:  relate(rs2,rs3)
08:  select(livro,rs3)
09:  relate(rs3,rs2)
10:  select(emprestimoLivro,rs4)
11:  relate(rs4,rs3)
12:  relate(rs1,rs4)
```

atualizações. Caso um processo tenha múltiplas operações relacionadas apenas com as variáveis de entrada, subprogramas distintos podem ser induzidos utilizando LGPDB para prover cada operação. A concatenação destes subprogramas resulta no programa final que modela todo o processo.

No intuito de analisar a viabilidade desta abordagem, foi elaborado um experimento simples em que o objetivo é modelar o processo de transferência de dinheiro entre duas contas correntes para ser usado por um sistema de gestão de operações financeiras. A Tabela 5.10 mostra o subconjunto de tabelas do banco de dados de um sistema hipotético, criado apenas para testar a indução desta funcionalidade. As tabelas e os atributos são auto-explicativos.

Tab. 5.10: Subconjunto de tabelas de um sistema de gestão de operações financeiras. Atributos em itálico são chaves para relacionar tabelas.

Tabela	**Campos**
cliente	*id*, nome, cpf, endereco, telefone
conta	*id*, numero, *agencia_id*, *cliente_id*, saldo
poupanca	*id*, numero, *agencia_id*, *clienet_id*, saldo
transacao	*id*, operacao, *conta_id*, valor, codigo

Definido o banco de dados, registros hipotéticos foram inseridos, seguindo a mesma metodologia do experimento apresentado na Seção 5.2. Na etapa seguinte, foi definida a funcionalidade **F1**: "Dadas as contas **C1** na agência **A1** e **C2** na agência **A2**, transferir a quantia **V** de **C1** para **C2**". Múltiplos casos de avaliação foram elaborados para induzir um programa para prover esta funcionalidade. A Tabela 5.11 mostra um exemplo de caso de avaliação. Neste exemplo, a entrada do programa é o conjunto de variáveis necessárias para a execução de todas operações e o resultado é um conjunto com quatro operações de banco de dados, sendo que cada operação é responsável por uma parte do processo.

No processo de geração de programa, esta tarefa é dividida automaticamente na indução de quatro

Tab. 5.11: Exemplo de caso de avaliação para a funcionalidade **F1**

.

Funcionalidade	F1
Entrada	999-1, 111-1, 999-2, 111-2, 500
Resultado	database.update(conta, “3002”, “saldo”, “1800”); database.update(conta, “3001”, “saldo”, “1500”); database.insert(transacao,{“4001”, send, “3002”, “500”, “t01”}); database.insert(transacao,{“4001”, recv, “3001”, “500”, “t01”});

subprogramas, cada um modelando uma operação de banco de dados. No caso da funcionalidade **F1**, serão necessários quatro subprogramas para a modelagem do processo completo. Apesar de cada subprograma modelar apenas uma parte da solução, todas as variáveis de entrada são acessíveis aos mesmos, independente da necessidade. Os comandos de banco de dados especificados nos casos de avaliação são separados. Cada programa é avaliado considerando a modificação causada por um dos comandos. Portanto, esta técnica está limitada na modelagem de processos que podem ser quebrados em etapas independentes. Os subprogramas deste experimento foram induzidos utilizando a mesma abordagem utilizada no experimento da seção 5.2. A solução final é a combinação dos quatro subprogramas para a formação de um único programa que modela todo o processo. Entre a concatenação de cada subprograma é adicionado o comando **clearEnvironment()** que reinicializa as variáveis do ambiente de execução para que os valores das variáveis utilizadas pelo subprograma anterior não afete as instruções do subprograma atual. O resultado final é apresentado na Figura 5.12. Uma vez que os subprogramas são induzidos separadamente e modelam processos parecidos, trechos de código idênticos aparecem em diferentes subprogramas. Portanto, o tamanho da solução final poderia ser reduzido, implementando um método de geração automática de funções.

Os resultados apresentados nesta seção são preliminares, mas já indicam a possibilidade de modelar processos mais complexos e comuns em problemas do mundo real.

5.5 Considerações Finais

Este capítulo apresentou uma proposta para indução de programas genéticos capazes de modelar processos de manipulação de informação. A ferramenta LGPDB foi implementada para validar a proposta em problemas hipotéticos de manipulação de dados. Trabalhos anteriores na área de manipulação de dados por programas genéticos focaram quase que exclusivamente na descoberta de conhecimento e extração de padrões de bases de dados. Neste último caso, o banco de dados é tratado como uma entidade estática, contendo conhecimento e padrões que podem ser extraídos. Portanto, programas genéticos não precisam ter a capacidade de alterar os registros para extrair essas informa-

```
//SubProg1
01: addRule(_,numero,equals,A2,rule2)
02: select(agencia,rs1)
03: filter_2(rs1,rule2)
04: Select(conta,rs2)
05: addRule(_,numero,equals,C2,rule1)
06: filter_2(rs2,rule1)
07: relate(rs2,rs1)
08: setValue(saldo,V,+,rs2)
//SubProg3
09: clearEnvironment()
10: addRule(_,numero,equals,A1,rule2)
11: addRule(_,numero,equals,C1,rule1)
12: select(agencia,rs4)
13: filter_2(rs4,rule2)
14: select(conta,rs3)
15: relate(rs3,rs4)
16: filter_2(rs3,rule1)
17: setValue(saldo,V,-,rs3)
//SubProg2
18: clearEnvironment()
19: addRule(_,numero,equals,B1,rule1)
20: addRule(_,numero,equals,A1,rule2)
21: select(conta,rs4)
22: select(agencia,rs2)
23: filter_2(rs4,rule2)
24: filter_2(rs2,rule1)
25: relate(rs4,rs2)
26: createRelation(transacao,rs2,rs4)
27: setValue(operacao,send,=,rs2)
29: setValue(valor,V,=,rs2)
30: setValue(id,transID,=,rs2)
31: //SubProg4
32: clearEnvironment()
33: addRule(_,numero,equals,A2,rule1)
34: addRule(_,numero,equals,B2,rule2)
35: select(agencia,rs3)
36: select(conta,rs4)
37: filter_2(rs4,rule1)
38: filter_2(rs3,rule2)
39: relate(rs4,rs3)
40: createRelation(transacao,rs1,rs4)
41: setValue(operacao,recv,=,rs1)
42: setValue(id,transID,=,rs1)
43: setValue(valor,V,=,rs1)
```

Fig. 5.12: Programa induzido para modelar a funcionalidade **F1**, resultante da concatenação de quatro subprogramas induzidos separadamente.

ções da base de dados. Por sua vez, no caso da modelagem de processos que normalmente alteram os registros armazenados no banco de dados, como é o caso da maioria dos processos presentes em sistemas de TI, é uma premissa prover tal capacidade às soluções candidatas. Programas induzidos por LGPDB são capazes de consultar, excluir, inserir e atualizar registros num banco de dados relacional. Os resultados indicam a viabilidade da automatização de algoritmos de manipulação de dados. A abordagem se mostrou capaz de modelar processos simples que ainda continuam sendo implementados por programadores humanos.

Além de possibilitar que programadores humanos se dediquem a tarefas mais complexas, LGPDB também cria a possibilidade que processos de manipulação de informação sejam modelados e implementados por profissionais que conhecem os processos, porém que não são programadores. Analistas, gerentes de produto, administradores de bancos de dados ou até mesmo usuários finais do sistema poderiam implementar alguns destes processos utilizando a ferramenta descrita neste capítulo.

Em algumas situações, o LGPDB também pode reduzir os custos de manutenção. No caso de alterações no banco de dados que não afetam os casos de avaliação como, por exemplo, a mudança de nome de atributos ou tabelas, basta realizar novamente o processo de geração automática de pro-

gramas para se obter os mesmos processos, só que modelados na nova versão do banco de dados. Em outros casos, o custo da alteração dos casos de avaliação para a atualização de um determinado processo pode ser menor do que o custo de se alterar um algoritmo em linguagem de programação para a modelagem do mesmo processo.

No caso de metodologias de desenvolvimento de software que já utilizam casos de avaliação, também chamados de casos de teste, para validação de software, estes mesmos casos poderiam ser reaproveitados para a geração automática de programas. Desta forma, o custo da criação dos casos de avaliação para geração automática de programas poderia ser reduzido ou eliminado.

Este trabalho apresenta um primeiro passo na geração automática de programas que modelam processos que alteram dados num banco de dados relacional. As diversas formas de utilização desta ferramenta, assim como a mensuração de seu benefício, devem ser explorados em maior profundidade em trabalhos futuros, conforme sugerido no Capítulo 6.

Capítulo 6

Conclusão

Soluções de software se tornaram uma ferramenta fundamental e de uso cotidiano na sociedade moderna. Atualmente, ferramentas desta modalidade são utilizadas para comunicação, organização pessoal, entretenimento, automação, dentre outras tantas aplicações. Uma modalidade específica de software, conhecida como sistemas de tecnologia de informação (TI), representa boa parte dos esforços atuais na área de desenvolvimento deste tipo de solução. Sistemas de TI, responsáveis pela modelagem de processos e suporte à tomada de decisão, estão presentes em praticamente todas as áreas de atuação humana e em atividades de diferentes níveis de complexidade. Soluções que empregam software para automação de processos podem ser encontradas tanto em aplicações relativamente simples, como gerenciamento de um pequeno consultório médico, quanto em aplicações mais complexas, como sistemas bancários.

Assim como o desenvolvimento de qualquer outro tipo de produto, o desenvolvimento de produtos de software necessita de alguma metodologia que divida o processo de desenvolvimento em etapas, da concepção à entrega do produto final. Diferentes metodologias, princípios e ferramentas foram propostos para auxiliar a execução destas etapas. Inclusive, algumas delas contam com métodos para automatização ou auxílio à tomada de decisão. Entretanto, dentre todas as etapas presentes na concepção de um produto de software, a elaboração de algoritmos e codificação provavelmente é a mais trabalhosa e com menor grau de automatização.

Este trabalho apresentou uma abordagem visando automatizar o desenvolvimento de algoritmos para modelagem de processos de manipulação de informação. O objetivo da ferramenta apresentada neste trabalho, o LGPDB, não é automatizar toda a implementação do sistema. Muitas tarefas de implementação de algoritmos ainda são um desafio para programadores humanos, providos de uma capacidade cognitiva que ainda não conseguimos prover a máquina alguma. Entretanto, tarefas mais simples e que ainda são executadas por programadores humanos poderiam ser automatizadas com o uso de ferramentas como o LGPDB, permitindo um direcionamento dos esforços de progra-

mação para tarefas mais desafiadoras. Este tipo de automatização também permite que pessoas que conheçam os processos a serem modelados, mas que não tenham conhecimento de programação de computador, possam gerar programas para modelar tais processos.

Os resultados obtidos com o LGPDB indicam que é viável a geração automática de programas capazes de manipular dados organizados num banco de dados relacional. Uma vez que estes programas conseguem alterar os dados, é possível modelar processos que alteram o estado do sistema, ao invés de apenas realizarem consultas. Os programas gerados nos experimentos apresentam boa interpretabilidade por humanos, fator que favorece a utilização da ferramenta num cenário em que profissionais humanos terão que interagir com a ferramenta. Por outro lado, dentre os pontos em que o LGPDB precisa melhorar está o fato de que o processo de indução é inerente às propriedades estatísticas dos dados armazenados no banco de dados, conforme descrito na Seção 5.3. Permitir a modelagem de processos com associações mais complexas é outra questão relevante, principalmente considerando a aplicação da abordagem em problemas do mundo real. Porém, é importante ressaltar que este é um trabalho inicial na linha de geração automática de programas para o desenvolvimento de sistemas de TI. Há diversas questões que precisam ser tratadas em trabalhos futuros, conforme ainda será discutido neste capítulo.

De forma mais específica, este trabalho apresentou as seguintes contribuições:

- Uma análise da evolução dos processos de desenvolvimento de software ao longo das últimas décadas e como abordagens cada vez mais sofisticadas têm sido empregadas para este propósito.

- Uma revisão das diferentes abordagens de programação genética e de algumas aplicações realizadas desde a proposição desta técnica.

- Na Seção 5.1, foi proposta uma arquitetura computacional, combinando um sistema de gerenciamento de bando de dados e um módulo de geração automática de programas, ambos com funcionalidades específicas para a geração de programas capazes de manipular informação num banco de dados relacional.

- Na Seção 5.1.2 foram propostas métricas de distância para avaliar programas candidatos a modelar tarefas de consulta, exclusão, inserção e atualização de registros num banco de dados.

- Na Seção 5.3, foi apresentado um método para incentivar a exploração de relacionamentos pelas soluções candidatas. Nas gerações iniciais do processo evolucionário, programas candidatos devem encontrar a sequência de relacionamentos correta para associar atributos de entrada com as tabelas correspondentes, permitindo assim o acesso aos atributos de interesse para modelar um determinado relacionamento.

A realização dos experimentos, apresentados no Capítulo 5, evidenciou algumas questões que devem ser tratadas em trabalhos futuros:

- Os experimentos mostraram que o LGPDB é capaz de gerar programas para modelagem de processos de manipulação de informação. Mesmo sabendo que existem processos de mesma complexidade em sistemas reais, seria interessante abordar problemas de maior complexidade. A identificação dos limites da ferramenta por meio de uma análise de escalabilidade pode auxiliar a proposta de melhorias.
- O espaço de busca de soluções pode ser reduzido pela implementação de operadores genéticos mais sofisticados. Na atual versão do LGPDB, o operador de mutação, por exemplo, sorteia um parâmetro de instrução aleatoriamente, independentemente dos parâmetros atuais da instrução. No caso da instrução **Relate(ResultSet rs1, ResultSet rs2)**, por exemplo, o sorteio de um **ResultSet** que não tem relacionamento com o **ResultSet** vinculado ao outro parâmetro não agrega nada ao programa candidato. Neste caso, só deveriam ser sorteados **ResultSets** que tenham relacionamento com o outro parâmetro da instrução. Portanto, as probabilidades de sorteio de parâmetros de instrução poderiam estar relacionadas aos parâmetros atuais da instrução manipulada.
- Com o objetivo de aproveitar melhor os recursos computacionais durante o processo evolucionário, seria interessante a implementação de mecanismos para permitir paralelismo, por exemplo, na etapa de avaliação. Uma vez que a avaliação de um indivíduo não afeta a de outros, a população de indivíduos a serem avaliados poderia ser dividida em múltiplos conjuntos, cada um avaliado num processador distinto.
- Os programas gerados por LGPDB utilizam instruções para manipular um banco de dados específico. Para viabilizar aplicações em problemas do mundo real, seria interessante a implementação do mesmo conjunto de instruções suportados por LGPDB, porém para manipular bancos de dados amplamente utilizados (e.g., PostgreSQL, MySQL).
- O desenvolvimento de um método para converter programas no formato do LGPDB para SQL permitiria a utilização do LGPDB na geração automática de transações SQL.
- Comparar os custos de atualização de um processo de manipulação de informação implementado por LGPDB e por uma linguagem de programação tradicional. Esta análise é importante pois atualmente praticamente todo projeto de software está em constante evolução. Sendo assim, o custo de atualização do sistema é um elemento fundamental para a definição de como e com quais tecnologias um dado sistemas será implementado.

Referências Bibliográficas

Albrecht, A. (1979). Measuring application development productivity, *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, Vol. 83, p. 92.

Albrecht, A. and Gaffney Jr, J. (1983). Software function, source lines of code, and development effort prediction: a software science validation, *Software Engineering, IEEE Transactions on* (6): 639–648.

Ambler, S. (2000). Mapping objects to relational databases: What you need to know and why, *IBM DeveloperWorks* .

Anderberg, M. R. (1973). *Cluster analysis for applications / Michael R. Anderberg*, Academic Press, New York.

Bagnall, A., Rayward-Smith, V. and Whittley, I. (2001). The next release problem, *Information and Software Technology* **43**(14): 883–890.

Balzer, R. (1985). A 15 year perspective on automatic programming, *Software Engineering, IEEE Transactions on* (11): 1257–1268.

Banzhaf, W. (1998). *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*, Morgan Kaufmann.

Beck, K. (2001). *Extreme programming explained: embrace change*, Addison-Wesley.

Beck, K. (2003). *Test-driven development: by example*, Addison-Wesley Professional.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R. et al. (2001). Manifesto for agile software development, *The Agile Alliance* pp. 2002–04.

Bhattacharya, M., Abraham, A. and Nath, B. (2001). A linear genetic programming approach for modeling electricity demand prediction in victoria, *Proceedings of the hybrid information systems,*

first international workshop on hybrid intelligent systems, Adelaide, Australia, Vol. 11, Springer-Verlag, p. 12.

Bichier, M. and Lin, K. (2006). Service-oriented computing, *Computer* **39**(3): 99–101.

Boden, E. and Martino, G. (1996). Testing software using order-based genetic algorithms, *Proceedings of the First Annual Conference on Genetic Programming*, MIT Press, pp. 461–466.

Boehm, B. (1988). A spiral model of software development and enhancement, *Computer* **21**(5): 61–72.

Bogner, J. and Bouzerdoum, A. (1997). The evolutionary pre-processor: Automatic feature extraction for supervised classification using genetic programming, *Proceedings of the second annual conference: July 13-16, 1997, Stanford University*, p. 304.

Brameier, M. and Banzhaf, W. (2001). A comparison of linear genetic programming and neural networks in medical data mining, *Evolutionary Computation, IEEE Transactions on* **5**(1): 17–26.

Brameier, M. and Banzhaf, W. (2007). *Linear genetic programming*, Springer-Verlag New York Inc.

Briand, L., Labiche, Y. and Shousha, M. (2005). Stress testing real-time systems with genetic algorithms, *Proceedings of the 2005 Conference on Genetic and Cvolutionary Computation*, ACM, pp. 1021–1028.

Burgess, C. and Lefley, M. (2001). Can genetic programming improve software effort estimation? a comparative evaluation, *Information and Software Technology* **43**(14): 863–873.

Chen, P. (1976). The entity-relationship model - toward a unified view of data, *ACM Transactions on Database Systems (TODS)* **1**(1): 9–36.

Cook, D. and Holder, L. (2007). *Mining graph data*, Wiley-Blackwell.

Cooper, K., Schielke, P. and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms, *ACM SIGPLAN Notices*, Vol. 34, ACM, pp. 1–9.

Cramer, N. (1985). A representation for the adaptive generation of simple sequential programs, *Proceedings of the First International Conference on Genetic Algorithms*, Vol. 183, p. 187.

Dain, R. (1998). Developing mobile robot wall-following algorithms using genetic programming, *Applied Intelligence* **8**(1): 33–41.

Darwin, C. (1859). On the origin of species by means of natural selection london, *J. Murray* .

De Falco, I., Della Cioppa, A. and Tarantino, E. (2002). Discovering interesting classification rules with genetic programming, *Applied Soft Computing* **1**(4): 257–269.

Dolado, J. (2000). A validation of the component-based method for software size estimation, *Software Engineering, IEEE Transactions on* **26**(10): 1006–1021.

Duda, R. O., Hart, P. E. and Stork, D. G. (2000). Pattern classification (2nd edition).

Eggermont, J., Kok, J. N. and Kosters, W. A. (2004). Genetic programming for data classification: partitioning the search space, *SAC*, p. 1001.

Fayad, M., Johnson, R. and Schmidt, D. (1999). Building application frameworks: object-oriented foundations of framework design.

Fayyad, U., Wierse, A. and Grinstein, G. (2002). *Information visualization in data mining and knowledge discovery*, Morgan Kaufmann Pub.

Fogel, D. (1997). The advantages of evolutionary computation, *Bio-Computing and Emergent Computation* pp. 1–11.

Fogel, L., Owens, A. and Walsh, M. (1966). Artificial intelligence through simulated evolution.

Freitas, A. (1997). A genetic programming framework for two data mining tasks: classification and generalized rule induction, *Genetic Programming 1997: Proc 2nd Annual Conf*, Morgan Kaufmann, pp. 96–101.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc.

Garcia-Molina, H. and Salem, K. (1992). Main memory database systems: An overview, *IEEE Transactions on Knowledge and Data Engineering* pp. 509–516.

Gillett, R., Kerr, A., Sallaberger, C., Maharaj, D., Martin, E., Richards, R. and Ulitsky, A. (2004). A hybrid range imaging system solution for in-flight space shuttle inspection, *Electrical and Computer Engineering, 2004. Canadian Conference on*, Vol. 4, pp. 2147–2150.

Gurbaxani, V. and Whang, S. (1991). The impact of information systems on organizations and markets, *Communications of the ACM* **34**(1): 59–73.

Han, J. and Fu, Y. (1994). Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases, *Proc. AAAI Workshop on Knowledge in Databases*, Vol. 94, pp. 157–168.

Han, J. and Kamber, M. (2006). *Data Mining: Concepts and Techniques*, The Morgan Kaufmann series in data management systems, Elsevier.

Harman, M. (2007). The current state and future of search based software engineering, *2007 Future of Software Engineering*, IEEE Computer Society, pp. 342–357.

Harman, M., Hierons, R. and Proctor, M. (2002). A new representation and crossover operator for search-based optimization of software modularization, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1351–1358.

Harman, M. and Jones, B. (2001). Search-based software engineering, *Information and Software Technology* **43**(14): 833–839.

Hedberg, S. (1997). Robots playing soccer? robocup poses a new set of ai research challenges, *IEEE Expert* **12**(5): 5–9.

Holland, J. (1975). Adaptation in Natural and Artificial Systems.

Kalra, R. and Deo, M. (2007). Genetic programming for retrieving missing information in wave records along the west coast of india, *Applied Ocean Research* **29**(3): 99–111.

Khoshgoftaar, T., Liu, Y. and Seliya, N. (2004). A multiobjective module-order model for software quality enhancement, *Evolutionary Computation, IEEE Transactions on* **8**(6): 593–608.

Koza, J. (1992). *Genetic programming: on the programming of computers by means of natural selection*, The MIT press.

Koza, J. (1994). Genetic programming II: automatic discovery of reusable programs.

Koza, J., Andre, D., Bennett III, F. and Keane, M. (1996). Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming, *Proceedings of the First Annual Conference on Genetic Programming*, MIT Press, pp. 132–140.

Koza, J., Bennett III, F., Andre, D. and Keane, M. (1996). Automated wywiwyg design of both the topology and component values of electrical circuits using genetic programming, *Proceedings of the First Annual Conference on Genetic Programming*, MIT Press, pp. 123–131.

Koza, J., Bennett III, F., Andre, D., Keane, M. and Dunlap, F. (1997). Automated synthesis of analog electrical circuits by means of genetic programming, *Evolutionary Computation, IEEE Transactions on* **1**(2): 109–128.

Krasner, G. and Pope, S. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80, *J. Object Oriented Program.* **1**(3): 26–49.

Langdon, W. and Poli, R. (1998). Fitness causes bloat: Mutation, *Genetic Programming* pp. 37–48.

Larman, C. (2004). *Agile and iterative development: a manager's guide*, Addison-Wesley Professional.

Luke, S. et al. (1998). Genetic programming produced competitive soccer softbot teams for robocup97, *Genetic Programming* pp. 214–222.

Mabu, S., Hirasawa, K. and Hu, J. (2007). A graph-based evolutionary algorithm: genetic network programming (gnp) and its extension using reinforcement learning, *Evolutionary Computation* **15**(3): 369–398.

Main, J. (1990). The winning organization, *Management information systems*, Scott, Foresman & Co., pp. 42–49.

March, S. and Smith, G. (1995). Design and natural science research on information technology, *Decision support systems* **15**(4): 251–266.

Maximilien, E. and Williams, L. (2003). Assessing test-driven development at ibm, *Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, pp. 564–569.

Montana, D. (1995). Strongly typed genetic programming, *Evolutionary Computation* **3**(2): 199–230.

Neumann, J. (1945). First draft of a report on the edvac, *University of Pennsylvania* .

Ng, W. and Ravishankar, C. (1995). Relational database compression using augmented vector quantization, *ICDE*, Published by the IEEE Computer Society, p. 540.

Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code, *Advances in genetic programming* pp. 311–331.

Nordin, P. and Banzhaf, W. (1997a). An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming, *Adaptive Behavior* **5**(2): 107.

Nordin, P. and Banzhaf, W. (1997b). Real time control of a khepera robot using genetic programming, *Control and Cybernetics* **26**: 533–562.

Parnas, D. and Clements, P. (1985). A rational design process: How and why to fake it, *Formal Methods and Software Development* pp. 80–100.

Pearson, K. (1901). Liii. on lines and planes of closest fit to systems of points in space, *Philosophical Magazine Series 6* **2**(11): 559–572.

Poli, R. (1997). Evolution of graph-like programs with parallel distributed genetic programming, *Genetic Algorithms: Proceedings of the Seventh International Conference*, Morgan Kaufmann, pp. 346–353.

Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*, 7th edn, McGraw-Hill, Inc., New York, NY, USA.

Quinlan, J. (1993). *C4. 5: programs for machine learning*, Morgan kaufmann.

Rechenberg, I. (1973). Evolutionsstrategie–optimierung technisher systeme nach prinzipien der biologischen evolution.

Royce, W. (1970). Managing the development of large software systems, *proceedings of IEEE WESCON*, Vol. 26, Los Angeles.

Ryu, T. and Eick, C. (1996a). Deriving queries from examples using genetic programming, *The Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pp. 303–306.

Ryu, T. and Eick, C. (1996b). MASSON: discovering commonalities in collection of objects using genetic programming, *Proceedings of the First Annual Conference on Genetic Programming*, MIT Press, pp. 200–208.

Salim, M. and Yao, X. (2002). Evolving sql queries for data mining, **2412**: 225–235.

Shirakawa, S., Ogino, S. and Nagao, T. (2007). Graph structured program evolution, *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM, pp. 1686–1693.

Siegwart, R. and Nourbakhsh, I. (2004). Autonomous mobile robots, *Massachusetts Institute of Technology* .

Silberschatz, A., Korth, H. and Sudarshan, S. (2002). *Database System Concepts*, Vol. 72, McGraw-Hill New York.

Sutherland, J. (1995). Business object design and implementation workshop, *ACM SIGPLAN OOPS Messenger*, Vol. 6, ACM, pp. 170–175.

Teller, A. (1996). Evolving programmers: The co-evolution of intelligent recombination operators, *Advances in Genetic Programming*, MIT Press, pp. 45–68.

Wang, S., Chen, Y. and Wu, P. (2009). Function sequence genetic programming, *Emerging Intelligent Computing Technology and Applications. With Aspects of Artificial Intelligence* pp. 984–992.

Williams, L., Kudrjavets, G. and Nagappan, N. (2009). On the effectiveness of unit test automation at microsoft, *20th International Symposium on Software Reliability Engineering*, IEEE, pp. 81–89.

Witten, I. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann.

Wolpert, D. and Macready, W. (1997). No free lunch theorems for optimization, *Evolutionary Computation, IEEE Transactions on* **1**(1): 67–82.