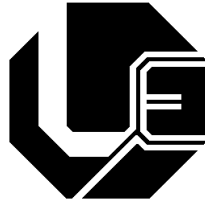


UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



**Programação genética paralela com
Pareto: uma ferramenta para
modelagem via regressão simbólica**

Leonardo Garcia Marques

Leonardo Garcia Marques

**Programação genética paralela com
Pareto: uma ferramenta para
modelagem via regressão simbólica**

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como requisito parcial para a obtenção do título de Mestre em Ciências.

Área de concentração: Processamento da Informação, Inteligência Artificial

Orientador: Keiji Yamanaka, Dr

Uberlândia
2013

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

- M357p
2013 Marques, Leonardo Garcia, 1983-
Programação genética paralela com Pareto: uma ferramenta para modelagem via regressão simbólica / Leonardo Garcia Marques. -- 2013.
111 f. : il.
- Orientador: Keiji Yamanaka.
- Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.
1. Engenharia elétrica - Teses. 2. Informática - Teses. 3. Programação paralela (Computação). -- Teses. 4. Inteligência artificial - Teses. 5. Programação genética (Computação). I. Marques, Leonardo Garcia. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

Leonardo Garcia Marques

**Programação genética paralela com
Pareto: uma ferramenta para
modelagem via regressão simbólica**

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como requisito parcial para a obtenção do título de Mestre em Ciências.

Área de concentração: Processamento da Informação, Inteligência Artificial

Uberlândia, 26 de Novembro de 2013

Banca Examinadora:

Keiji Yamanaka, Dr – FEELT/UFU

Alexsandro S. Soares, Dr – FACOM/UFU

Wesley Pacheco Calixto, Dr – NExT/IFG

Sérgio A. A. de Freitas, Dr – FGA/UnB

À minha família e aos meus amigos.

Agradecimentos

À minha família, em especial à minha mãe Dorgeni e minha à irmã Adriana. Ninguém sabe mais sobre as o meu trajeto do que elas.

Às minhas irmãs Maria do Carmo e Mariuza, que tantas vezes e tão agradavelmente me receberam em suas casas em Uberlândia.

Ao meu orientador, Prof. Keiji Yamanaka, que tanta confiança me creditou nestes anos de orientação. Sua presteza em me atender e sua dedicação foram fundamentais.

Aos companheiros de laboratório: Igor Peretta, Mônica Sakuray, Ricardo Boaventura, Gerson Flávio; bons amigos cuja clareza de pensamentos tanto me ajudou em frutíferas conversas que tivemos;

Aos amigos de Itumbiara: Hugo Xavier, Gesmar Júnior, Ghunter Viajante e Wellington do Prado, companheiros de viagem e de rotina da pós-graduação; Evoney Queiroz, Eduardo Mizael, Roberta Ponciano, Jucélio Araújo e tantos outros bons amigos que sempre me apoiaram.

Aos amigos que fiz no período que passei na Reitoria do IFG: Roberval Lustosa, Cristiano Domingues, Saulo Rodrigues, Ricardo Moreira, Douglas Santana, Renan Oliveira, Viviane Gomes, Wagner Bento, Édio Cardoso, Luciano Eduardo e Júlio Mota.

Aos velhos amigos dos tempos da graduação, presentes ainda hoje: Marcos Bueno com quem compartilho as inquietações da vida acadêmica e que tantas vezes me ofereceu estadia em Uberlândia; Alex Araújo, que sempre acreditou mais na minha capacidade do que eu mesmo.

Ao Prof. Wesley Pacheco, pela amizade e por apontar ótimos caminhos para a conclusão do meu trabalho.

Ao Programa de Pós-Graduação da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia, em especial aos Professores Alexandre Cardoso e Edgard Lamounier e à Cinara Fagundes, pelo apoio, orientação e incentivo.

“Existe efetiva grandiosidade neste modo de encarar a Vida que, juntamente com todas as suas diversas capacidades, teria sido insuflada numas poucas formas, ou talvez numa única, e que, enquanto este planeta continua a girar, obedecendo à imutável Lei da Gravidade, as formas mais belas, mais maravilhosas, evoluíram a partir de um início tão simples e ainda prosseguem hoje em dia neste desenvolvimento.”

(Charles Darwin – A Origem das Espécies)

Resumo

Marques, L. G. **Programação genética paralela com Pareto: uma ferramenta para modelagem via regressão simbólica.** 111 p. Dissertação – Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia, 2013 .

Indução de programas envolve a descoberta de programas de computador que produzem alguma saída desejada quando estes são submetidos a alguma entrada em particular. Um exemplo é a regressão simbólica, ferramenta de modelagem que busca expressões de funções matemáticas para ajustar determinado conjunto de dados multivariados, mapeando variáveis de entrada para variáveis de saída de controle. A programação genética, uma sub-área da computação evolutiva que usa analogia da teoria da evolução de Darwin e algumas ideias de genética, é uma técnica automática para produzir programas de computador amplamente usada para resolver problemas. No entanto, a implementação da programação genética não é trivial para a maioria dos profissionais, além de demandar alto poder computacional. Este trabalho apresenta uma implementação paralela de programação genética simples de se manusear, otimizada para computadores de arquitetura com múltiplos núcleos e que satisfaz o critério competitivo de simplicidade estrutural e exatidão na predição, através de variação especial multiobjetiva de programação genética, chamada programação genética com Pareto. A implementação proposta tem ganhos de desempenho proporcionais à quantidade de núcleos disponíveis em uso, além de ter sido aplicada com sucesso em diversos tipos de problemas de regressão.

Palavras-chave: Programação genética. Processadores *multicore*. Dominância de Pareto.

Abstract

Marques, L. G. **Parallel Pareto Genetic Programming: a tool to modeling via symbolic regression.** 111 p. Master Thesis – Faculty of Electrical Engineering, Federal University of Uberlândia, 2013 .

Program induction involves the inductive discovery of a computer program that produces some desired output when presented with some particular input. An example is the symbolic regression, a modeling tool that seeks mathematical expressions of functions to fit a given multivariate data set, mapping input variables to output variables of control. The genetic programming, a subarea of evolutive computing that uses an analogy of Darwin's evolutionary theory and some ideas from the genetics field, is an automatic technique for producing a computer program widely used to solve such problems. However, implementing genetic programming is not trivial for most professionals, besides demanding high computational power. This work presents a parallel implementation of genetic programming simple to handle, optimized for computers with multicore architecture, and satisfying competitive criteria of structural simplicity model and prediction accurate model, through a special multi-objective flavor of a genetic programming, called Pareto Genetic Programming. The proposed implementation has performance gains proportional to the amount of available cores in use, and has been successfully applied to several types of regression problems.

Keywords: Genetic Programming. Multicore processors. Pareto dominance.

Lista de ilustrações

Figura 1 – Exemplo de código LISP e sua árvore correspondente.	24
Figura 2 – Árvore da expressão $(* (+ (- 2 x) (* 2 x)) (- x 2))$	24
Figura 3 – Obtenção do resultado da expressão $(((9 - y) * (y * y)) * (x - (-1)))$ por meio do caminhamento em pré-ordem.	25
Figura 4 – Cromossomo tipicamente utilizado em GP Linear.	26
Figura 5 – Esquema de representação linear proposto por Banzhaf (1993).	27
Figura 6 – A árvore de uma expressão e seu grafo correspondente.	27
Figura 7 – Processo de criação de uma árvore.	30
Figura 8 – Árvore gerada pelo método <i>full</i>	31
Figura 9 – Árvore gerada pelo método <i>grow</i>	31
Figura 10 – Exemplo de torneio com $k = 3$ aplicado a um problema de maximização.	35
Figura 11 – <i>Crossover</i> entre dois indivíduos.	37
Figura 12 – <i>Crossover</i> de um ponto em duas árvores de mesmo formato.	39
Figura 13 – <i>Crossover</i> de um ponto em duas árvores de formatos diferentes.	39
Figura 14 – <i>Crossover</i> uniforme em duas árvores de mesmo formato.	40
Figura 15 – <i>Crossover</i> uniforme em duas árvores de formatos diferentes.	40
Figura 16 – Coordenadas marcadas em uma árvore para <i>crossover</i> de preservação de contexto.	41
Figura 17 – Lógica de escolha de pontos no <i>crossover</i> de preservação de contexto.	41
Figura 18 – Crescimento do tamanho médio das árvores que compõem uma popu- lação.	45
Figura 19 – Conjunto de soluções que tentam minimizar dois objetivos dispostos nos eixos x e y	46
Figura 20 – Cálculo do <i>crowding-distance</i>	50
Figura 21 – Representação do procedimento do NSGA-II.	53
Figura 22 – Crescimento do tamanho médio da população para a função $2x^3 - 5x + 8$	58

Figura 23 –Crescimento do tamanho médio da população para a função $6 \sin x + \cos y$.	58
Figura 24 –Crescimento do tamanho médio da população para a função $x^4 + x^3 + x^2 + x$.	59
Figura 25 –Valor do erro do melhor indivíduo de cada geração para a função $2x^3 - 5x + 8$.	59
Figura 26 –Valor do erro do melhor indivíduo de cada geração para a função $6 \sin x + \cos y$.	60
Figura 27 –Valor do erro do melhor indivíduo de cada geração para a função $x^4 + x^3 + x^2 + x$.	60
Figura 28 –Topologia de migração em anel.	64
Figura 29 –Unidade central de processamento.	66
Figura 30 –Gráfico comparativo de eficiência entre a versão paralela de GP e a versão sequencial.	69
Figura 31 –Gráfico de comparação.	78
Figura 32 –Gráfico do tamanho médio da árvores.	79
Figura 33 –Buscar arquivo com padrões de treinamento.	82
Figura 34 –Tela de configuração de parâmetros.	83
Figura 35 –Exibição dos resultados.	84
Figura 36 –Aba de exibição de gráficos: comparação.	85
Figura 37 –Aba de exibição de gráficos: tamanho médio das árvores da população.	86
Figura 38 –Multiplexador de 6 entradas modelado pela ferramenta PPGP.	91
Figura 39 –Aproximação do lançamento oblíquo.	93
Figura 40 –Lançamento oblíquo.	94
Figura 41 –Aproximação da função dupla exponencial.	95
Figura 42 –Organização das classes e interfaces que representam uma árvore.	109

Lista de tabelas

Tabela 1 – Terminais comumente utilizadas em GP.	28
Tabela 2 – Funções comumente utilizadas em GP.	28
Tabela 3 – Exemplo de arquivo com padrões de treinamento.	73
Tabela 4 – Primitivas disponíveis.	74
Tabela 5 – Valores de parâmetro para a Equação 20.	95

Lista de algoritmos

1	Algoritmo Genético Básico	22
2	Algoritmo Básico de Programação Genética	23
3	Algoritmo para avaliar expressões simbólicas.	26
4	Algoritmo para criação de árvores.	30
5	Procedimento <i>fast-non-dominated-sort</i>	49
6	Procedimento <i>crowding-distance-assignment</i>	51
7	Algoritmo genético com NSGA-II	52
8	Lógica de funcionamento do <i>framework fork/join</i>	67

Lista de siglas

API *Application Programming Interface*

CPU *Central Processing Unit*

GA *Genetic Algorithms*

GP *Genetic Programming*

GPU *Graphics Processing Unit*

NSGA *Non-dominated Sorting Genetic Algorithm*

NSGA-II *Non-dominated Sorting Genetic Algorithm II*

PPGP *Parallel Pareto Genetic Programming*

PTC1 *Probabilistic Tree Creation 1*

PTC2 *Probabilistic Tree Creation 2*

SCPC *Strong Context Preserving Crossover*

SPEA *Strength Pareto Evolutionary Algorithm*

SPEA2 *Strength Pareto Evolutionary Algorithm 2*

WCPC *Weak Context Preserving Crossover*

Sumário

1	Introdução	16
1.1	Objetivos	17
1.2	Metodologia	17
1.3	Organização da dissertação	18
2	Programação genética	20
2.1	Computação evolutiva: dos algoritmos genéticos à programação genética	20
2.2	Representações	22
2.2.1	GP baseados em árvores	23
2.2.2	Outras representações	26
2.3	Definições dos conjuntos de primitivas	28
2.4	Geração da população inicial	29
2.5	Função de aptidão	31
2.6	Seleção	33
2.6.1	Roleta viciada	33
2.6.2	Seleção por <i>ranking</i>	33
2.6.3	Torneio	34
2.7	Operadores genéticos	35
2.7.1	Reprodução	36
2.7.2	Cruzamento	36
2.7.3	Mutação	40
2.7.4	Edição	42
2.7.5	Permutação	43
2.8	A escolha de parâmetros	43
3	Redução do efeito <i>bloat</i> via dominância de Pareto	44
3.1	Algoritmos evolutivos multiobjetivos	45
3.2	NSGA-II	48

3.2.1	<i>Fast non-dominated sort</i>	48
3.2.2	<i>Crowding-distance</i>	49
3.2.3	O <i>loop</i> principal	51
3.3	Programação genética com Pareto	53
3.3.1	Implementação	54
3.4	Resultados e discussões	56
4	Programação genética paralela em processadores <i>multicore</i>	61
4.1	A alta demanda por poder de processamento da GP	61
4.2	Abordagem paralela	62
4.3	O modelo de ilhas	63
4.4	Modelo de ilhas em programação genética com Pareto	64
4.5	Processadores <i>multicore</i> e programação concorrente	65
4.5.1	Arquitetura básica de computadores	65
4.5.2	<i>Threads</i> e <i>Hyper-Threading</i>	65
4.5.3	Tecnologia <i>multicore</i>	66
4.5.4	O <i>framework fork/join</i>	67
4.6	Implementação	67
4.7	Resultados e discussões	69
5	Uma ferramenta para programação genética paralela com Pareto	71
5.1	A ferramenta para desenvolvedores	71
5.1.1	Conjunto de amostras	72
5.1.2	Primitivas	73
5.1.3	Operadores genéticos disponíveis	76
5.1.4	Tipos de função de aptidão	77
5.1.5	Gráficos disponíveis	78
5.1.6	Parâmetros padrão	79
5.1.7	Exemplo	80
5.2	A ferramenta como aplicativo	81
5.2.1	Aba Dados	81
5.2.2	Aba Parâmetros	81
5.2.3	Aba Resultados	83
5.2.4	Aba Gráficos	84
5.3	Resultados e discussões	84
6	Regressão simbólica via programação genética	87
6.1	Funções polinomiais com uma variável	88
6.1.1	Polinômio de grau 2: $x^2 + x + 1$	88
6.1.2	Polinômio de grau 3: $x^3 + x^2 + x + 1$	88

6.1.3	Polinômio de grau 4: $x^4 + x^3 + x^2 + x$	88
6.2	Funções polinomiais com duas variáveis	88
6.2.1	Polinômio $x^2 + y^2 + 1$	88
6.3	Funções com logaritmos	89
6.3.1	Função $5 * \log(x) + x + 1$	89
6.4	Funções trigonométricas com uma variável	89
6.4.1	Função $(\sin x)/x$	89
6.4.2	Função $\sin x + \cos x$	89
6.4.3	Função $\sin^2 x + \cos x$	89
6.5	Funções trigonométricas com duas variáveis	90
6.5.1	Função $\sin^2 x + \cos y$	90
6.5.2	Observações	90
6.6	Reescrevendo equações	90
6.7	Síntese de circuitos digitais combinacionais	91
6.8	Modelagem do lançamento oblíquo no vácuo	92
6.9	Aproximação da função dupla exponencial para medir descargas elétricas atmosféricas	94
Conclusão		96
Referências		98
 Apêndices		105
APÊNDICE A Exemplificação do <i>framework fork/join</i>		106
APÊNDICE B Detalhes na implementação do PPGP		108
B.1	Diagrama de classe da árvore	108
B.2	O uso do <i>framework fork/join</i>	108

Introdução

A teoria da evolução das espécies, proposta por Charles Darwin, diz que todas as complexas estruturas biológicas hoje existentes provêm do longo processo de evolução, composto por recombinação de material genético (reprodução sexual) e mutação, ambos direcionados à melhor adaptação ao meio. Em sua base, todas essas estruturas são formadas por componentes simples (genes) que, quando combinados de forma adequada, manifestam uma ou outra característica.

Koza (1992b) argumenta que os programas de computadores estão entre os artefatos mais complexos criados pelo homem e questiona se há a possibilidade de os programas evoluírem e “se criarem”. O próprio Koza responde esta pergunta com o paradigma de programação genética, técnica de Computação Evolutiva que trabalha com indução de programas capazes de evoluírem (por meio da aplicação de operadores genéticos) e gerarem respostas desejadas, sempre que submetidos a entradas específicas.

Ainda de acordo com Koza (1992b), a indução de programa pode ser aplicada a diferentes tipos de problemas e a terminologia empregada adapta-se ao uso. Assim, um programa de computador pode ser uma fórmula, um plano, uma estratégia de jogo ou de controle, um procedimento computacional, um modelo, um projeto, uma árvore de decisão, uma expressão matemática, uma sequência de operações, etc. As entradas desses programas podem corresponder a valores obtidos via sensores, variáveis de estado, variáveis independentes, atributos, sinais de entrada, variáveis conhecidas ou argumentos de funções. Já as saídas podem ser tomadas como variáveis dependentes, variáveis de controle, instruções de controle e decisão, ações, movimentações, sinais de saída, variáveis desconhecidas ou retorno de funções.

Infelizmente, os algoritmos de computação evolutiva são, de maneira geral, de difícil implementação. A programação genética, em especial, trabalha com a manipulação de complexas estruturas de dados. Embora esta técnica possa, potencialmente, ser utilizada por diversos profissionais das mais variadas áreas, seu uso acaba restrito àqueles que possuem familiaridade não apenas com os conceitos de computação evolutiva, mas também com técnicas avançadas de programação, como manipulação de complexas estruturas de

dados.

1.1 Objetivos

Figura como objetivo primário dessa dissertação o desenvolvimento de uma ferramenta de apoio ao uso de programação genética que possa ser utilizado tanto por profissionais com conhecimento em programação e em computação evolutiva (para utilizá-la com API) quanto por profissionais não versados nessas duas áreas (na forma de uma aplicação) e que almejam aproveitar os benefícios da programação genética mantendo o foco no domínio do problema. Os usuários desta ferramenta poderão tratar problemas de modelagem via regressão simbólica.

Para alcançar o objetivo principal, alguns objetivos secundários fizeram-se necessários:

- ❑ Conceituar e discutir os diversos aspectos da programação genética;
- ❑ Melhorar a qualidade das respostas dadas pelos sistemas de programação genética, obtendo estruturas mais simples e com erro pequeno ou zero;
- ❑ Melhorar o desempenho da programação genética por meio de técnicas de paralelismo.

Há diversas opções, livres ou proprietárias, voltadas para este fim – como a *framework JGAP* (<http://jgap.sourceforge.net/>), *EpochX* (<http://www.epochx.org>) e *toolbox* de *softwares* como *Matlab*. Porém, como será visto ao longo da dissertação, a ferramenta desenvolvida no presente trabalho traz reunidos conceitos que, simultaneamente, não aparecem nas demais, como o aproveitamento otimizado e transparente dos processadores *multicore* e a aplicação de técnicas multiobjetivos.

1.2 Metodologia

Os conceitos de programação genética serão apresentados por meio de revisão bibliográfica, essencial ao entendimento dos assuntos tratados ao longo do texto. Seu algoritmo básico é apresentado, assim como diversos detalhes inerentes à sua implementação.

A melhoria da qualidade das respostas obtidas pelo sistemas de programação genética é conseguida por meio do combate ao *bloat*, efeito que acrescenta grande complexidade estrutural aos indivíduos da população sem que haja o beneficiamento em termos de aptidão. Este combate é feito por meio da técnica conhecida por programação genética com Pareto – abordagem multiobjetiva capaz de conduzir a evolução por regiões do espaço de busca onde se encontram os indivíduos com menor complexidade estrutural e com erro próximo (ou igual) a zero.

A melhoria do desempenho na execução da programação genética é conseguida com a utilização do *framework fork/join*, presente na linguagem Java a partir de sua sétima edição. Este *framework* divide a carga de processamento entre os núcleos presentes no computador em uso, possibilitando, assim, a implementação do consagrado modelo de ilhas.

Os conceitos apresentados são a base para a construção de uma ferramenta multi-plataforma que traz implementados os conceitos de paralelismo e dominância de Pareto. Há também a possibilidade de estender suas funcionalidades, pois o usuário pode criar novas funções – próprias do seu domínio de aplicação – e utilizá-las como novos genes na população.

Por fim, são apresentados diversos resultados obtidos pela ferramenta desenvolvida quando utilizada em problemas de regressão simbólica.

1.3 Organização da dissertação

Esta dissertação está organizada a partir dos aspectos teóricos mais primordiais da programação genética e segue em direção à aplicação por parte dos usuários, discutindo importantes conceitos durante este caminho.

As principais conceitualizações teóricas necessárias à compreensão dos temas tratados nesta dissertação estão contidos no Capítulo 2. Este capítulo apresenta o algoritmo básico da programação genética e discute sobre os diversos caminhos possíveis para implementá-lo.

O Capítulo 3 define o efeito *bloat* e detalha como este pode ser combatido por meio da programação genética com Pareto, uma variação da técnica original apresentada no Capítulo 2. A programação genética com Pareto utiliza conceitos de computação evolutiva multiobjetivo baseada em dominância de Pareto para gerar respostas mais compactas. O *Non-dominated Sorting Genetic Algorithm II* (NSGA-II), utilizado em algoritmos genéticos multiobjetivos, é visto em detalhes neste capítulo, que é finalizado com testes e comparações com a abordagem clássica.

O Capítulo 4 discute como a tecnologia de processadores com múltiplos núcleos de processamento pode ser utilizada para aumentar o desempenho da programação genética e como isso pode ser feito de maneira transparente ao usuário. Este capítulo discorre sobre a arquitetura *multicore*, apresenta o modelo de ilhas (uma das principais abordagens de paralelização dadas para algoritmos evolutivos) e os recursos disponíveis nas linguagens de programação modernas para a distribuição da carga de processamento pelos núcleos disponíveis.

O Capítulo 5 apresenta a PPGP, uma ferramenta voltada para modelagem via regressão simbólica que foi implementada com base em todos os conceitos discutidos nos capítulos anteriores. A PPGP pode ser utilizada tanto como uma API, por profissionais

com avançados conhecimentos em programação e computação evolutiva, quanto como um aplicativo – para profissionais não versados nas duas áreas de conhecimento citadas, mas que querem se beneficiar com o potencial da programação genética.

O Capítulo 6 traz a conceitualização de regressão simbólica, problema adequadamente tratado com programação genética, e apresenta diversas possibilidades de aplicação.

Programação genética

A Programação Genética – ou *Genetic Programming* (GP) – é uma técnica que automaticamente produz programas para solucionar dados problemas de maneira exata ou aproximada (KOZA, 2003). Koza (1990a) registrou sua patente, mas no trabalho de Grings (2006) verifica-se que há registros anteriores de pesquisadores que, assim como Koza, tendo como referência os algoritmos genéticos, construíram modelos capazes de evoluir programas, tanto em linguagem LISP quanto em representações de árvores e *strings*. Dentre eles, destacam-se Fujiko e Dickinson (1987), Hicklin (1986) e Cramer (1985).

Esta técnica apresenta notável potencial de aplicações. Koza (2003) argumenta que o fato de muitos programas poderem ser representados como problemas de busca faz com que grande quantidade de variados tipos de problemas (como os de controle, classificação, sistemas de identificação ou projeto) possam ser resolvidos via programação genética. A área de projeto, em especial, é uma fonte de problemas desafiadores, já que a GP pode automaticamente produzir resultados competitivos nesta área, que requer criatividade e inteligência humana.

Este capítulo apresenta uma visão geral sobre a programação genética, dissertando rapidamente sobre sua relação com os algoritmos genéticos e se concentrando nos detalhes inerentes à sua implementação.

2.1 Computação evolutiva: dos algoritmos genéticos à programação genética

A programação genética é uma técnica que tem suas bases nos algoritmos genéticos – ou *Genetic Algorithms* (GA), desenvolvidos por Holland (1975), que, por sua vez, é antecedida pelos conceitos apresentados por Turing (1950), que apresentou as noções gerais do que foi chamado de “busca genética” ou “busca evolutiva” (GRINGS, 2006). Ao conjunto de técnicas computacionais surgidas a partir da década de 1960 e baseadas nos processos naturais de genética e evolução, foi dado o nome de *Computação Evolutiva*,

havendo destaque para:

- ❑ Programação Evolutiva;
- ❑ Estratégias Evolutivas;
- ❑ Algoritmos Genéticos;
- ❑ Programação Genética.

A programação evolutiva é uma técnica proposta por Fogel (1962) cujo objetivo é evoluir máquinas de estado – autômatos. Já a técnica de estratégias evolutivas foi apresentada por Bäck, Hammel e Schwefel (1997) e evolui um único indivíduo por meio de mutação.

Devido à sua forte correlação com a GP, uma análise detalhada da técnica de algoritmos genéticos se faz necessária. Os Algoritmos Genéticos são algoritmos de busca baseados em mecanismos de seleção natural e genética (GOLDBERG, 1989). Esta técnica utiliza seleção natural/evolução e genética para obter resultados em problemas de busca e otimização e foi a base para a criação da programação genética – tanto que ambas compartilham o mesmo algoritmo básico. Conforme o observado por Linden (2008), quando se trabalha com GA, alguns conceitos e termos do campo da genética são adaptados ao meio computacional:

- ❑ Um **indivíduo** corresponde a uma **solução**;
- ❑ Um **cromossomo**, a **representação** dessa solução (em geral, uma cadeia de caracteres representando alguma informação relativa às variáveis do problema);
- ❑ A **reprodução sexual** se dá por meio do **operador de cruzamento**, que combina partes de dois indivíduos (pais) de maneira a formar outros indivíduos;
- ❑ A **mutação** é obtida pela **modificação aleatória** de um ou mais elementos da cadeia de caracteres;
- ❑ A **população** é um **conjunto de soluções em potencial**;
- ❑ O meio **ambiente** no qual ela se desenvolve corresponde ao **problema**;
- ❑ Uma **geração** consiste em um **ciclo de execução** do algoritmo.

Os algoritmos genéticos são modelos computacionais utilizados em problemas de otimização que não possuem algoritmos determinísticos eficientes. Eles atuam em um espaço de busca – região onde se encontram as possíveis soluções para o problema – e são heurísticas aplicadas no tratamento de problemas da classe NP-Difíceis, cujo tempo de execução pode demandar mais processamento do que é atualmente possível com as arquiteturas

dos computadores atuais (CORMEN et al., 2001). Eles são utilizados para otimizar uma função de aptidão que dá uma espécie de pontuação ao indivíduo dentro do ambiente.

Todo o desenvolvimento é baseado na Teoria da Evolução das Espécies, de Darwin. Indivíduos diferentes (mais ou menos adaptados ao ambiente) compõem uma população e têm a capacidade de combinar-se (cruzamentos ou reprodução sexual) entre si, gerando novos indivíduos com características herdadas de ambos. De acordo com a teoria, há um favorecimento aos indivíduos melhor adaptados e espera-se que estes tenham melhores condições de sobreviver e deixar descendentes. Com o passar das gerações, as condições ambientais (eficácia na resolução do problema) vão naturalmente selecionando os mais aptos, obtendo uma população “melhorada”. Além dos cruzamentos, outro fator que diferencia os indivíduos da população ao longo das gerações é a mutação. Esta ocorre em percentuais menores e proporciona modificações que podem ser mais ou menos benéficas à adaptação do indivíduo ao meio.

O funcionamento de um algoritmo genético, em sua versão mais básica, é mostrado no Algoritmo 1, adaptado de Linden (2008). Nele, é possível ver os conceitos de genética aplicados à solução de um problema.

Algoritmo 1 Algoritmo Genético Básico

- 1: Inicialize a população
 - 2: **enquanto** a condição de parada não é atingida **faça**
 - 3: Calcule a aptidão de cada indivíduo da população
 - 4: Selecione os pais
 - 5: Execute o cruzamento
 - 6: Execute a mutação
 - 7: Avalie os resultados
 - 8: Selecione os sobreviventes para compor a nova geração
 - 9: **fim enquanto**
-

A condição de parada, que figura na linha 2 do Algoritmo 1, pode ser implementada de diferentes maneiras. A abordagem mais comum é associá-la à quantidade máxima de gerações que a população pode evoluir.

O principal fator que difere os algoritmos genéticos da programação genética é a maneira de representar a solução: enquanto um algoritmo genético utiliza *strings* de tamanho fixo, a programação genética baseia-se em genótipos de tamanhos variados, o que facilita a criação de novas estruturas (ARAÚJO, 2004). Embora essas particularidades reflitam na maneira como os operadores genéticos são implementados, a sequência de passos necessária à programação genética assemelha-se bastante ao seu antecessor, como pode ser visto no Algoritmo 2, adaptado de Poli, Langdon e McPhee (2008).

Mesmo com a semelhança nos passos de execução, a GP apresenta alta complexidade de implementação. As próximas seções expõem seus principais conceitos, na tentativa de elucidar e detalhar suas características mais significativas.

Algoritmo 2 Algoritmo Básico de Programação Genética

- 1: Crie randomicamente uma população inicial de programas com as primitivas disponíveis
 - 2: **repita**
 - 3: Execute cada programa e acerte sua aptidão
 - 4: Selecione um ou dois programas da população com uma probabilidade baseada na aptidão
 - 5: Crie um novo programa aplicando operadores genéticos
 - 6: **até** uma solução aceitável ser encontrada ou alguma condição de parada for atingida.
 - 7: **retorne** o melhor indivíduo da população.
-

2.2 Representações

Há variadas maneiras de representar os indivíduos de uma população a ser evoluída por programação genética e todas utilizam estruturas de dados cujo tamanho e formato sejam variáveis. Esta seção apresenta esquemas de GP baseados em árvores, listas e grafos.

2.2.1 GP baseados em árvores

Uma árvore T é composta por um *conjunto* de *nós* e estes interagem entre si em um relacionamento do tipo *pai-filho*. Caso esse conjunto seja não vazio, haverá um nó especial, denominado *raiz*, que não possui pai. Nos demais casos, cada nó v de T possui um único pai w (GOODRICH; TAMASSIA, 2003). Os nós que não possuem filhos são chamados *terminais* ou *folhas*, enquanto os demais recebem o nome de *não-terminais*. Neste contexto, há também o conceito de *profundidade* d (do inglês *depth*) de um nó, que equivale à distância (quantidade de arestas) que o separa da raiz, que possui profundidade 0. O maior valor de d em uma árvore equivale à sua profundidade.

A importância dessa estrutura de dados reside no fato de que as primeiras implementações de programação genética realizadas por Koza foram feitas representando os indivíduos em forma de árvores, que são naturalmente implementadas na linguagem LISP, utilizada por ele. Nesta linguagem, os programas possuem somente a forma sintática, não havendo diferenciação entre dado e programa, o que facilita na manipulação genética da população (ARAÚJO, 2004). Um programa em LISP pode ser mapeado diretamente em forma de árvore – uma árvore sintática. A Figura 1 apresenta um código em LISP juntamente com sua árvore correspondente.

Qualquer expressão matemática pode ser transcrita em LISP: basta converter sua representação para a notação prefixa (GRAHAM, 1996). A Expressão (2), por exemplo, é a Expressão (1) reescrita na forma prefixa. Como pode ser visto na Figura 2, essa notação traz facilidades à compreensão do relacionamento de uma (sub)expressão e sua (sub)árvore correspondente (POLI; LANGDON; MCPHEE, 2008).

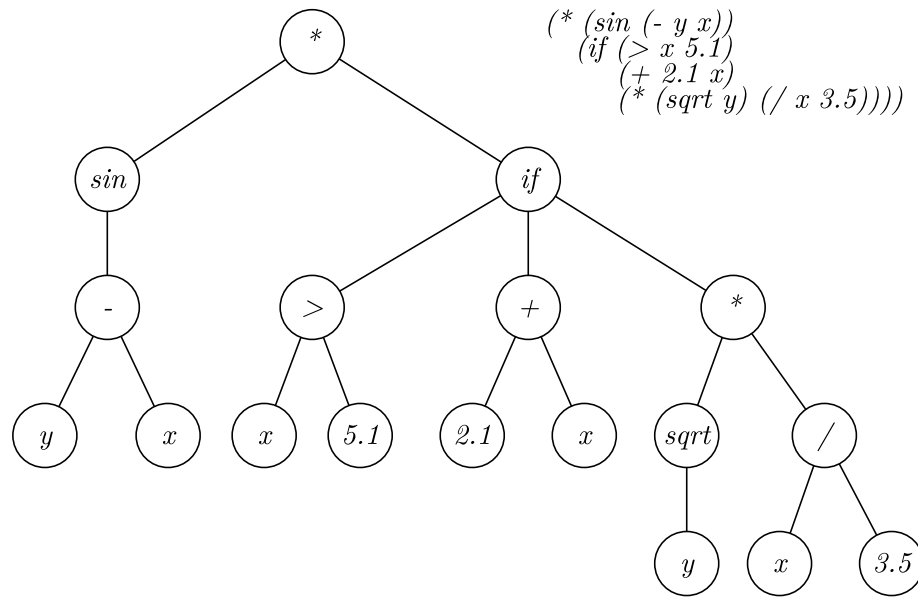
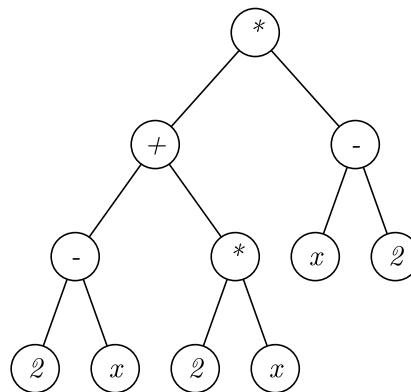


Figura 1 – Exemplo de código LISP e sua árvore correspondente.

$$(((2 - x) + (2 * x)) * (x - 2)) \quad (1)$$

$$(* (+ (- 2 x) (* 2 x)) (- x 2)) \quad (2)$$

Figura 2 – Árvore da expressão $(* (+ (- 2 x) (* 2 x)) (- x 2))$.

Nas árvores utilizadas em programação genética, os nós não-terminais correspondem às funções (aritméticas, por exemplo) e a quantidade de filhos desses nós representa o número de argumentos da função – também chamado de *aridade*. Já os nós terminais podem corresponder a constantes, variáveis ou funções sem argumento.

A programação dessa estrutura de dados é, em geral, realizada por meio de manipulação de grande número de ponteiros – ou referências, dependendo do contexto. Isso costuma ser um gargalo, visto que a alocação dinâmica de memória e seu correto referenciamento demandam alto esforço computacional. De fato, a facilidade de implementação de uma árvore está diretamente ligada à linguagem de programação escolhida. Em LISP, por exemplo, a implementação é direta, já que, conforme foi visto, a própria organização do código tende a comportar-se como uma árvore. A implementação apresentada neste trabalho foi realizada em linguagem Java, por razões que serão discutidas no Capítulo 5.

O resultado do programa (ou da expressão) contido na árvore é obtido executando seus nós em uma ordem que preserve as regras de precedência subjacentes e isso é conseguido por meio do *caminhamento em pré-ordem* (POLI; LANGDON; MCPHEE, 2008). Neste tipo de caminhamento, o nó raiz w da árvore T é visitado primeiro e, em seguida, percorre-se recursivamente as sub-árvores, cujas raízes são filhos de w . Quando um terminal é alcançado, seu valor é mensurado, servindo de argumento para os nós superiores no retorno da recursão. Ao findar do processo, w retornará o valor avaliado pela árvore, conforme exemplificado na Figura 3. Dentre as abordagens possíveis, é a forma mais eficiente de caminhamento, com tempo execução na ordem $O(n)$, onde n é o número de nós (GOODRICH; TAMASSIA, 2003).

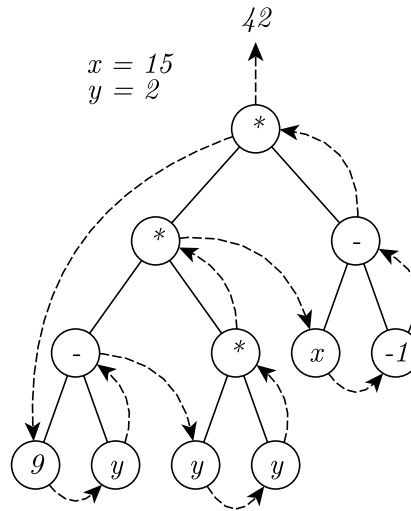


Figura 3 – Obtenção do resultado da expressão $((9 - y) * (y * y)) * (x - (-1))$ por meio do caminhamento em pré-ordem. Para $x = 15$ e $y = 2$, o resultado é 42.

O procedimento apresentado na Figura 3 é equivalente à avaliação de uma expressão prefixada escrita sob a forma de lista. Para avaliar uma expressão assim, considera-se o primeiro argumento como raiz da árvore e realiza-se a avaliação de maneira recursiva, tal qual é mostrado no Algoritmo 3. Tal procedimento é amplamente utilizado no tratamento simbólico de expressões.

Algoritmo 3 *avalia*(*expr*)

```

1: se expr é uma função então
2:   proc ← expr(1)  (O primeiro elemento equivale à raiz)
3:   valor ← proc(avalia(expr(2)), avalia(expr(3)), ...)  (Avalia seus argumentos)
4: senão
5:   se expr é uma variável or expr é uma constante então
6:     valor ← expr  (Obtém seu valor)
7:   senão
8:     valor ← expr()  (Função sem-argumentos: execute)
9:   fim se
10: fim se
11: retorne valor

```

2.2.2 Outras representações

As árvores não são as únicas estruturas de dados utilizadas em programação genética. Alguns problemas adequam-se melhor a outras formas de representação, como a linear e a em grafo.

2.2.2.1 Linear

A representação linear faz uso da maneira como um programa é organizado: uma sequência de instruções que são executadas sequencialmente, de cima para baixo, da esquerda para a direita. Embora linguagens do paradigma funcional possam ser diretamente mapeadas em árvores (como ocorre em LISP), a implementação de programas em outros tipos de linguagens trazem a necessidade de um interpretador que transforme a árvore em um programa pronto para ser avaliado pelo sistema. A representação linear faz uso desta característica para produzir programas que possam ser avaliados diretamente pelo compilador/interpretador da linguagem em uso e sua representação é dada pela Figura 4.

Instrução 1	Instrução 2	...	Instrução N
-------------	-------------	-----	-------------

Figura 4 – Cromossomo tipicamente utilizado em GP Linear. As instruções são dispostas sequencialmente e sua avaliação ocorre de maneira semelhante ao que é feito na interpretação de linguagens de programação. Adaptado de Poli, Langdon e McPhee (2008)

Banzhaf (1993) foi um dos primeiros a utilizarem essa abordagem. Seu método consiste em produzir e evoluir *strings* binárias que correspondem a códigos de programação. No momento da avaliação, a *string* binária é “traduzida” para o código correspondente e o programa é avaliado. A Figura 5 apresenta este esquema.

Perkis (1994) propôs um sistema de programação genética cujos programas, ao longo das gerações, são executados em uma máquina virtual baseada em pilha. Quando comparado com a implementação tradicional que utiliza árvores sintáticas, este método apresenta

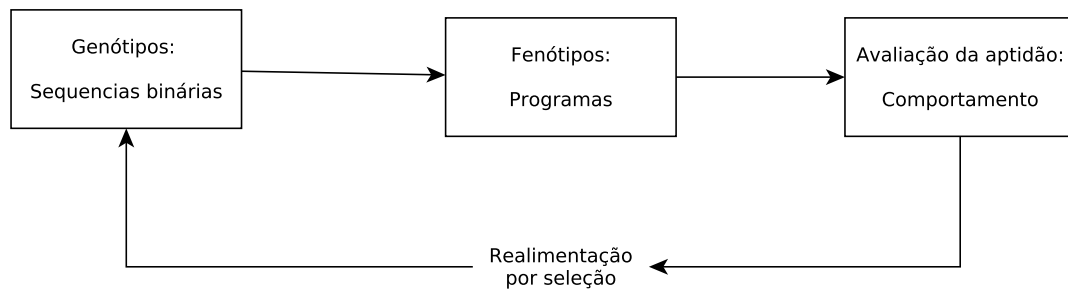


Figura 5 – Esquema proposto por Banzhaf (1993). É um esquema cíclico, composto por genótipo, fenótipos e avaliação. As sequências binárias são transformadas em programas e executadas, para medir a aptidão.

algumas vantagens concernentes à eficiência e à simplicidade de programação, chegando a ser mais eficiente para problemas de regressão simbólica e mapeamento de funções booleanas.

2.2.2.2 Grafos

Um grafo consiste em um conjunto de vértices, um conjunto de arestas e uma relação que os conecta. Uma árvore é, na verdade, um tipo particular de grafo, caracterizado por ser totalmente conectado e não possuir ciclos (FOULDS, 1991). Os grafos representam outra possibilidade para mapeamento de programas a serem evoluídos por GP.

Poli (1996) partiu dos conceitos de processamento paralelo e distribuído (RUMELHART; MCCLELLAND; GROUP, 1986) para representar programas como grafos direcionados e mostrou como esse método pode produzir programas mais compactos, quando comparados às suas versões correspondentes em árvores. Isso ocorre porque o grafo possibilita reunir em uma única região partes (sub-árvore) iguais que aparecem em diversos locais na árvore. A Figura 6 exemplifica este processo.

2.3 Definições dos conjuntos de primitivas

As árvores (ou demais estruturas de dados) utilizadas em programação genética têm a capacidade de representar estruturas altamente complexas. Porém, todas são compostas por elementos-base, que compõem os *conjuntos de primitivas*, que são específicos para cada área de aplicação. Tais conjuntos são separados em funções e terminais.

Os terminais são divididos em três categorias: as variáveis, as funções sem argumento e as constantes. As variáveis correspondem às entradas externas ao programa – x e y , por exemplo. As funções sem argumento, ou funções de aridade zero, são aquelas que apenas retornam um valor sempre que chamadas – a `rand()`, por exemplo – e as que modificam valores globais quando da sua execução. As constantes podem ser especificadas

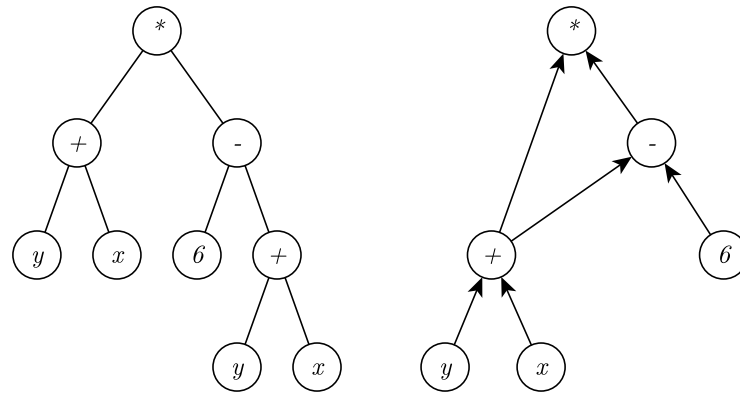


Figura 6 – A árvore de uma expressão e seu grafo correspondente. Com o grafo, as sub-árvores formadas por $(+ x y)$, que aparecem em dois locais, são reunidas, tornando a figura mais compacta.

no momento da criação das árvores que farão parte da população inicial. É comum a utilização da função `rand()` nesse momento, por exemplo. A tabela 1, adaptada de Poli, Langdon e McPhee (2008), oferece exemplos de terminais.

Tabela 1 – Terminais comumente utilizadas em GP.

Tipos de Primitivas	Exemplos
Variáveis	x, y
Valores constantes	3.14, 100
Funções sem argumento	<code>rand</code>

Já as funções, ou não-terminais, estão intrinsecamente ligadas à natureza do problema. Em um problema de síntese de circuitos digitais, por exemplo, as funções serão portas lógicas como *AND*, *OR* e *NOT*. A Tabela 2, adaptada de Poli, Langdon e McPhee (2008), exemplifica algumas funções comumente usadas em árvores de programação genética.

Tabela 2 – Funções comumente utilizadas em GP.

Tipos de Primitivas	Exemplos
Aritméticas	$+, *, -, /$
Matemáticas	$\sin, \cos, \tan$, etc.
Booleanas	<i>AND, OR, NOT</i>
Condicional	<i>if-then-else</i>
Laços	<i>for, repetat</i>

Além de identificar quais funções serão utilizadas para compor o conjunto, há a necessidade de se considerar o fato de que cada função deve aceitar como argumento qualquer valor válido gerado pelas combinações possíveis de outras funções, constantes e valores assumidos pelas variáveis. Esta propriedade recebe o nome de *fechamento*. Koza (1992b)

faz ampla discussão sobre esse tema e esta foi estendida por Poli, Langdon e McPhee (2008) e suas principais apontamentos são descritos nessa seção.

Embora pareça uma tarefa complexa, a garantia da propriedade de fechamento pode ser obtida acrescentando algumas restrições nas definições de funções. Basta tratar as restrições na própria implementação da função, como é exemplificado a seguir:

- ❑ Uma *divisão por zero* pode ser evitada implementando a *divisão protegida*, geralmente denotada por `%`. Neste operador, sempre que o valor zero for encontrado no denominador, será retornado o valor 1. Nos demais casos, o quociente é calculado normalmente;
- ❑ Um número *negativo* no argumento de uma função que calcula a *raiz quadrada* pode ser tratado calculando o *valor absoluto* do argumento antes de aplicá-lo à função. O mesmo vale para a função *logaritmo*, com a ressalva de retornar zero sempre que o argumento for zero.
- ❑ Se um valor *numérico* for aplicado em uma função que espera receber um valor *lógico*, pode-se adotar a convenção de considerar `false` qualquer valor negativo e `true` caso contrário.

Após garantir o fechamento dos conjuntos de terminais e funções, é necessário tratar também da *suficiência* desses conjuntos, que consiste na capacidade de expressar todas as respostas possíveis ao problema utilizando apenas as primitivas disponíveis. Mais especificamente, os conjuntos são suficientes se a combinação de todas as composições recursivas inclui pelo menos uma solução (POLI; LANGDON; MCPHEE, 2008).

Um conjunto formado pelas funções $\{AND, NOT\}$, por exemplo, é capaz de formar as combinações necessárias para resolver o problema de um circuito digital combinacional para verificar paridade. Porém, um conjunto de funções formado por $\{*, +, /, -\}$ não é capaz de representar a função e^x .

2.4 Geração da população inicial

A população inicial a ser evoluída por programação genética é um conjunto de expressões simbólicas composto pelas primitivas escolhidas para a resolução de um problema. Os apontamentos concernentes ao processo de geração da população inicial feitos por Koza (1992b) servem de base para a maioria das implementações de GP e seus principais conceitos são dados a seguir.

A criação de um indivíduo, como pode ser visto na Figura 7, inicia-se com o “sorteio” de uma função que representará a raiz da árvore. Seus argumentos são recursivamente sorteados dentro dos conjuntos de terminais e funções e segue até ser finalizado com a escolha de nós terminais.

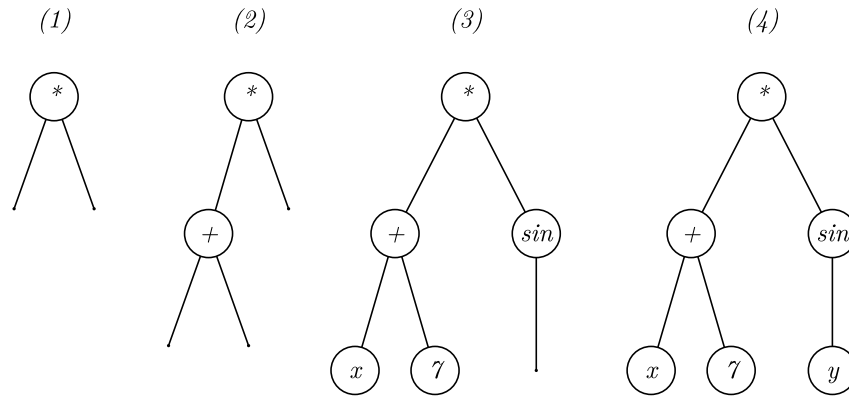


Figura 7 – Processo de criação de um árvore: (1) criação da árvore com a escolha da função $*$, com dois argumentos, como raiz; (2) escolha da função $+$, com dois argumentos, para ser um argumento de $*$; (3) escolha dos terminais x e 7 para argumentos de $+$ e a função $\sin$, com um argumento, para argumento de $*$; (4) escolha do terminal y para argumento de $\sin$.

O que determina a forma, o tamanho e a profundidade da árvore é a maneira como os nós são escolhidos. Os métodos mais antigos – e mais amplamente usados – são o *full* e *grow* e o *ramped half-and-half*, que é uma combinação dos outros dois.

O método *full* gera árvores cujas folhas possuem, todas, a mesma profundidade d . Isto é conseguido fazendo com que, na formação da árvore, sejam escolhidos apenas nós função até a árvore atingir uma profundidade $d - 1$, finalizando processo com a escolha de terminais. Já o *grow* diferencia-se do *full* por escolher qualquer tipo de nó (terminal ou função) até a árvore atingir uma profundidade máxima d . O Algoritmo 4 apresenta o processo recursivo de criação de árvores por esses dois métodos. Já nas Figuras 8 e 9, exemplos de árvores criadas com a utilização do *full* e do *grow*, respectivamente, podem ser vistos.

Algoritmo 4 `gen_rnd_expr(funcoes_set, terminais_set, max_d, metodo)`

```

1: se max_d = 0 or (metodo = grow and rand() <  $\frac{|\text{terminais\_set}|}{|\text{terminais\_set}| + |\text{funcoes\_set}|}$ )
   então
2:   expr  $\leftarrow$  escolha_elemento_randomicamente(terminais_set)
3: senão
4:   func  $\leftarrow$  escolha_elemento_randomicamente(funcoes_set)
5:   para i  $\leftarrow$  1 to aridade(func) faça
6:     arg_i  $\leftarrow$  gen_rnd_expr(funcoes_set, terminais_set, max_d - 1, metodo)
7:   fim para
8:   expr  $\leftarrow$  (func, arg_1, arg_2, ...)
9: fim se
10: retorne expr

```

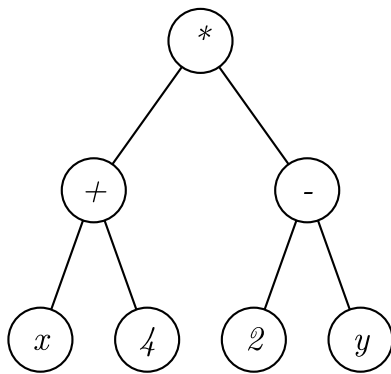


Figura 8 – Árvore gerada pelo método *full*.

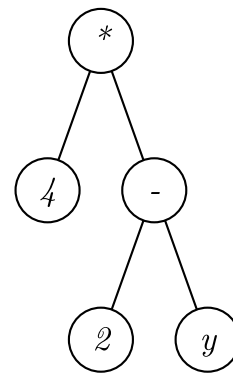


Figura 9 – Árvore gerada pelo método *grow*.

O *ramped half-and-half* é uma combinação dos dois métodos anteriores: 50% dos indivíduos são criados pelo método *grow* e o restante pelo método *full*. Para Koza (1992b), este é o método que apresenta os melhores resultados para uma ampla quantidade de problemas, sendo particularmente útil nas situações em que não se sabe (ou não se quer estabelecer a priori) o formato das árvores que farão parte da população.

Luke (2000b) oferece outras duas opções para a criação de árvores: o *Probabilistic Tree Creation 1* (PTC1) e *Probabilistic Tree Creation 2* (PTC2), que procuram não gerar distribuições completamente uniformes nos formatos das árvores. Eles garantem algo que os seus antecessores não podiam: probabilidade definida pelo usuário para a aparência de funções dentro das árvores, realizado com baixo esforço computacional.

O PTC1 é uma modificação do *grow* e permite ao usuário definir as probabilidades de aparecer funções nas árvores e um tamanho esperado para elas. O algoritmo garante que, na média, todas tenham tamanhos próximos, não garantindo, porém, essa variação. Com o PTC2, além das probabilidades, o tamanho, com pouca variação para mais, é garantido de maneira mais precisa do que o PTC1.

2.5 Função de aptidão

A melhor (ou pior) capacidade de adaptação de um indivíduo ao ambiente em que vive é um dos fatores apontados na teoria da Seleção Natural que influenciam na perpetuação da espécie: os mais aptos têm maiores chances de se reproduzir e, assim, passar suas características (seus genes) para as próximas gerações. Analogamente, em uma população de programas de GP, aquele que melhor reproduz um comportamento desejável é considerado mais apto e seus esquemas tendem a serem passados às próximas gerações. Quando um valor numérico é utilizado para medir a capacidade de adaptação, este recebe o nome de aptidão ou *fitness* (KOZA, 1992a).

Na maioria das vezes, a função de aptidão é calculada tomando por base um conjunto

de casos de teste representados por relações de entrada e saída. Sempre que possível, recomenda-se a utilização desse conjunto em sua totalidade, como é o caso do problema de avaliação de funções booleanas. Porém, para certos problemas este conjunto pode ser demasiadamente grande (ou até mesmo infinito), fazendo com que um subconjunto pequeno, mas suficientemente grande para representar o problema como um todo, seja adotado (KOZA, 1992b).

No momento da criação da população inicial, cada indivíduo é avaliado e a ele é atribuído o valor de aptidão. Este processo se repete ao longo das gerações, sempre antes da aplicação dos operadores genéticos. Este valor é calculado a partir da comparação entre as saídas produzidas pelo indivíduo e as saídas desejadas para cada caso de teste.

A aptidão bruta, cuja interpretação é inteiramente dependente do problema em análise, é o mais simples entre os vários tipos de funções de aptidão existentes. Seu valor é o somatório das diferenças absolutas entre os valores encontrados e os respectivos valores esperados. Matematicamente, a aptidão f_i do indivíduo i é dada pela Equação (3):

$$f_i = \sum_{j=1}^{N_c} |p_{ij} - s_j|, \quad (3)$$

com N_c representando o número de casos de teste, p_{ij} a saída do programa i para o caso de teste j e s_j a saída esperada para o caso j . Uma alternativa à Equação (3), amplamente usada em algoritmos de treinamento de Redes Neurais, é o Erro Quadrático Total (FAUSETT, 1994), dado pela Equação (4):

$$f_i = \sum_{j=1}^{N_c} (p_{ij} - s_j)^2, \quad (4)$$

Para problemas em que o melhor indivíduo é o que produz um valor numérico pequeno – como otimização de custos ou de erros – há a opção de *aptidão padronizada*, que mapeia os valores produzidos pela aptidão bruta para valores entre 0 e infinito (GRINGS, 2006). Isso pode ser conseguido por meio da adição (ou subtração) de uma constante.

Para enfatizar as pequenas diferenças numéricas entre as aptidões em uma população, este valor pode ser mapeado para o intervalo real $[0, 1]$. Esse método recebe o nome de *aptidão ajustada* e pode ser obtido pela fórmula:

$$a_i = \frac{1}{1 + s_i}. \quad (5)$$

O valor de aptidão pode conduzir a equívocos no momento da seleção dos indivíduos. Esse efeito é condicionado à abordagem utilizada na seleção, conforme será visto na Seção 2.6. De maneira geral, ele pode ser evitado com a *normalização da aptidão* n_i , obtido a partir do valor a_j dado pelo ajuste da aptidão:

$$n_i = \frac{a_i}{\sum_{j=1}^{N_p} a_j}. \quad (6)$$

2.6 Seleção

Em computação evolutiva, a forma como um indivíduo é escolhido para ser submetido aos operadores genéticos é também de fundamental importância. Métodos que garantam o beneficiamento do mais apto devem ser implementados de maneira eficaz e eficiente. Esta seção discute três desses métodos: a roleta viciada, o *ranking* e o torneio. Ambos são amplamente utilizados tanto em sistemas de programação genética quanto em algoritmos genéticos.

2.6.1 Roleta viciada

Este método de seleção baseia-se na proporcionalidade da aptidão. O nome “roleta viciada” faz alusão ao seu funcionamento: uma roleta é dividida em fatias proporcionais à aptidões dos indivíduos, de forma que os mais aptos recebem fatias maiores do círculo, o que aumenta as chances de eles serem sorteados, acontecendo exatamente o contrário com os menos aptos, mas sem impossibilitar por completo a escolha desses (GRINGS, 2006). Na prática, é uma escolha aleatória, porém influenciada pela magnitude da aptidão. Sua explicação, baseada na descrição dada por Tomassini e Calcolo (1995), é apresentada a seguir.

Inicialmente, calcula-se o total S resultante da soma das aptidões f_i de cada indivíduo da população com N indivíduos:

$$S = \sum_{i=1}^N f_i \quad (7)$$

e calcula-se a probabilidade p_i de cada indivíduo, que corresponde à proporção da aptidão em relação a S :

$$p_i = \frac{f_i}{S}, \quad i = \{1, 2, \dots, N\}. \quad (8)$$

Com essas informações, a probabilidade acumulada para cada indivíduo é calculada:

$$c_i = \sum_{k=1}^i p_k, \quad i = \{1, 2, \dots, N\}. \quad (9)$$

Para selecionar um indivíduo, escolhe-se um valor real $r \in [0, 1]$ para identificar o i -ésimo indivíduo da população, de maneira que $c_{i-1} < r \leq c_i$. Quando $r < c_1$, o primeiro indivíduo é selecionado.

Para ilustrar essa ideia, considere que $p_1 = 0,30$, $p_2 = 0,20$, $p_3 = 0,40$ e $p_4 = 0,10$. Então, temos $c_1 = 0,30$, $c_2 = 0,50$, $c_3 = 0,90$ e $c_4 = 1,00$. Se $r = 0,25$, então o primeiro indivíduo é selecionado, já que $r < c_1$. Porém, se $r = 0,96$, então o indivíduo 4 é selecionado, já que $c_3 < 0,96 \leq c_4$.

2.6.2 Seleção por *ranking*

O uso da roleta viciada torna-se problemático sempre que um indivíduo da população possui aptidão desproporcionalmente superior a todos os demais. Nessa situação, se diz que este indivíduo domina a população, pois ele quase sempre será selecionado, implicando na predominância de seus genes na maioria dos indivíduos já em poucas gerações. O resultado disso é a convergência prematura da população e a dificuldade da solução escapar de mínimos locais.

Uma das maneiras mais utilizadas para prevenir essa situação é a seleção por *ranking*, que mantém a pressão seletiva no mesmo nível em todas as gerações, independentemente do grau de convergência que a população já tenha tido (LINDEN, 2008). Esta técnica não utiliza diretamente o valor da função de aptidão: antes, é criado um *ranking* dos melhores indivíduos por meio do ordenamento da população de acordo com os valores dados pela função de aptidão.

Para evitar o gargalo trazido pelo constante processo de ordenamento, que ocorre em cada geração, é necessário escolher algoritmos eficientes. Os melhores algoritmos de ordenação conhecidos têm um tempo de execução $O(n \log n)$ (CORMEN et al., 2001). Estando a população ordenada em ordem crescente, o próximo passo é adotar novos valores para a função de avaliação. Tipicamente, estes valores são dados linearmente, muitas vezes se utilizando de funções como a exposta na Equação 10.

$$E(i, t) = Min + (Max - Min) \frac{rank(i, t) - 1}{N - 1}, \quad (10)$$

onde: Min é o valor dado ao indivíduo que estiver em pior colocação no ranking; Max é o valor dado ao indivíduo que estiver em melhor colocação; N corresponde ao número de indivíduos da população em análise; $rank(i, t)$ corresponde à colocação que o indivíduo i , numa geração t está no ranking.

Com os novos valores de avaliação, conseguidos após o ordenamento e com a aplicação da Equação 10 (ou outra, de mesma finalidade), é possível adotar um método de seleção comum, como o da roleta viciada. Verifica-se que, na média, um indivíduo que ocupa o lugar de número $N/2$ encontra-se exatamente entre os indivíduos de melhor e pior colocação, fazendo com que sua avaliação seja igual à média das avaliações, garantindo-lhe uma chance em N de ser selecionado (LINDEN, 2008).

2.6.3 Torneio

O torneio é o método de seleção mais utilizado em implementações de programação genética desde a sua concepção, por Koza (1990a), e consiste em escolher randomicamente k indivíduos e fazer com que eles compitam entre si. Vence o torneio aquele que possuir a melhor avaliação na função de aptidão (BLICKLE; THIELE, 1995).

A Figura 10 exemplifica a execução de um torneio com $k = 3$ sob uma população de oito indivíduos para um problema de maximização. Nesta figura, constata-se a existência de três indivíduos que certamente dominariam completamente a população na geração seguinte, com uma probabilidade de aproximadamente 49,9% de serem selecionados para os cruzamentos. Com o uso do torneio, essa probabilidade cai para cerca de 38,1%.

Indivíduo	Aptidão		<u>x_1</u>	x_7	x_8
x_1	200		x_2	<u>x_3</u>	x_5
x_2	100		<u>x_6</u>	x_4	x_4
x_3	9500		x_2	x_7	<u>x_1</u>
x_4	100	$\Rightarrow$	<u>x_5</u>	x_5	x_5
x_5	100		<u>x_3</u>	x_4	x_2
x_6	10000		x_4	x_2	<u>x_6</u>
x_7	1		x_4	<u>x_6</u>	x_5
x_8	40				

Figura 10 – Exemplo de torneio com $k = 3$ aplicado a um problema de maximização. À esquerda, os indivíduos da população e suas respectivas avaliações. Os vencedores de cada torneio (sublinhados) poderão efetuar o cruzamento. Adaptado de Linden (2008).

Como todos os integrantes do grupo que compõe o torneio são escolhidos aleatoriamente e todos com a mesma probabilidade de serem escolhidos, é possível que os melhores indivíduos (que certamente dominariam a próxima geração, no caso do uso da roleta viciada) não sejam escolhidos para competir. No caso da existência de um “super indivíduo”, este não dominará a população, por ter chances iguais de aparecer no torneio. Entretanto, sempre que ele participar, vencerá.

2.7 Operadores genéticos

Para que a Teoria da Seleção Natural, base da computação evolutiva, possa ser aplicada em um ambiente computacional, há a necessidade de um mapeamento de vários conceitos desta teoria em técnicas de programação. Nisto se baseiam os operadores genéticos.

Tecnicamente, os operadores genéticos são métodos usados para modificar as estruturas presentes na população a ser evoluída, oferecendo maneiras de promover suas adaptações. Koza (1992b) os divide em dois grupos: o primário, composto pela reprodução e pelo cruzamento (*crossover*) e o secundário, composto pelos operadores de mutação, permutação e edição.

A seguir, cada operador é apresentado e discussões acerca das suas diversas possibilidades de implementação são realizadas.

2.7.1 Reprodução

A reprodução traduz um dos conceitos mais conhecidos da teoria da evolução: a sobrevivência do mais apto. É um mecanismo assexuado, agindo sob um único indivíduo por vez e que, quando executado, produz um único elemento filho – ou *offspring*.

A forma como esse elemento é escolhido depende da abordagem utilizada, que é, em geral, alguma das abordagens descritas na Seção 2.6. Escolhido o elemento, basta copiá-lo para a população que formará a próxima geração.

Além de atender a um dos princípios da seleção natural, a reprodução tem também uma vantagem em termos de desempenho computacional. Como grande parte do poder de processamento utilizado em sistemas de programação genética é dedicado à avaliação da aptidão do indivíduo, a reprodução permite que este procedimento não seja necessário, por este valor já ser conhecido. Se a reprodução ocorrer com uma frequência de 10%, por exemplo, o sistema será 10% mais eficaz (KOZA, 1992b)

2.7.2 Cruzamento

O cruzamento (também chamado *crossover* ou recombinação) é um operador de reprodução sexuada que produz descendentes a partir da combinação do material genético de dois ou mais indivíduos – dependendo apenas da abordagem adotada.

O tipo mais comum é o chamado *crossover de sub-árvore*. Dados dois indivíduos independentemente selecionados (por algum método de seleção descrito na Seção 2.6, por exemplo), escolhe-se um ponto (um nó) em cada um. Da primeira árvore, descarta-se a sub-árvore cuja raiz é o nó selecionado, substituindo-a pela sub-árvore cuja raiz é o nó selecionado na segunda árvore. O restante da segunda árvore é descartado. Este processo pode ser visto na Figura 11.

Há em Poli, Langdon e McPhee (2008) a recomendação de que as partes do indivíduo envolvidas com o *crossover* sejam efetivamente copiadas, para não causar qualquer dano aos indivíduos originais. De fato, embora a cópia possa trazer uma sobrecarga em termos de processamento computacional, sua utilização previne muitos erros comumente cometidos quando se manipula ponteiros e referências.

Concernente ao ponto de *crossover* em cada árvore, há também uma recomendação. A probabilidade de escolha entre nós função costuma ser mantida em 90%, sendo os outros 10% deixados para escolha entre nós terminais.

2.7.2.1 *Crossover* homólogo

O *crossover* utilizado na programação genética não necessariamente segue todos os conceitos biológicos e isso é facilmente observado no *crossover* de sub-árvore. Na natureza, os cromossomos são combinados de maneira que os genes dos filhos se localizem aproximadamente nas mesmas posições dos genes dos pais, já que as cadeias de DNA

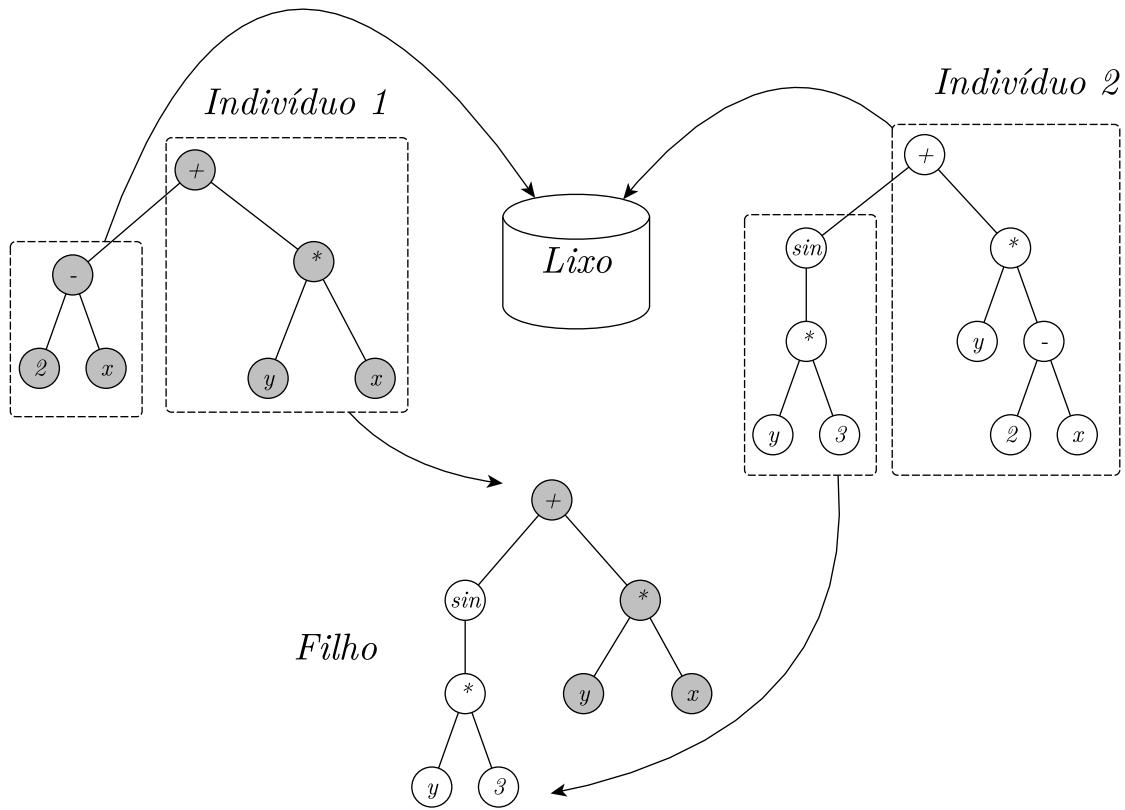


Figura 11 – *Crossover* entre os indivíduos $((2 - x) + (y * x))$ e $((\sin(y * 3)) + (y * (2 - x)))$, que resultou no filho $((\sin(y * 3)) + (y * x))$.

são devidamente alinhadas com genes de funcionalidades compatíveis. O cruzamento em árvores, por outro lado, é capaz de gerar descendentes com formatos diferentes de ambos os pais e isso pode ser prejudicial para a prole: um trecho de código (sub-árvore) “bom” pode gerar resultados medianos ou ruins quando movido de forma puramente aleatória para outros contextos em outras árvores (O'REILLY; OPPACHER, 1994). Além disso, o crescimento descontrolado do programa sem o devido beneficiamento da aptidão (conhecido como efeito *bloat*) também expõe as complicações que este tipo de *crossover* pode incorrer (LANGDON, 2000).

A preservação das posições dos genes pode ser conseguida com modificações no operador, que recebe o nome de *crossover homólogo* ou *recombinação homóloga*. Yamamoto e Tschudin (2005) utilizaram este conceito para evoluir protocolos de comunicação de redes de computadores, em uma aplicação em que a população inicial era formada por programas que correspondiam a “boas” respostas ou respostas aproximadas para o problema em análise. Como a recombinação homóloga consegue manter as propriedades do programa (ou seja, o contexto), foi possível realizar transformações, uma geração após a outra, gerando proles com funcionalidades similares, mas implementadas por diferentes caminhos e sem perder as vantagens que uma população inicial de qualidade pode trazer.

O conceito de recombinação homóloga foi implementado de diferentes maneiras, levando-se em consideração não somente as posições, mas também os formatos, os tamanhos e as profundidades das árvores envolvidas. Suas principais vertentes são apresentadas a seguir.

2.7.2.2 *Crossover* de um ponto

Poli e Langdon (1998) propuseram um cromossomo cuja ideia é semelhante à utilizada em algoritmos genéticos: o *crossover* de um ponto, em que os pontos de recombinação são escolhidos dentro das regiões de mesma forma nas duas árvores. Para identificar as regiões, basta percorrer as duas árvores paralelamente, iniciando na raiz de cada uma, e identificar os nós de mesma aridade. O processo é interrompido no momento em que as aridades diferirem. A Figura 12 ilustra um *crossover* entre dois indivíduos cujas formas das árvores são inteiramente iguais, onde qualquer nó da pode servir como ponto de recombinação. Já a Figura 13 ilustra o caso em que as árvores têm formatos diferentes, sendo preciso, primeiramente, identificar a área em comum entre elas que, no caso, está representada pelo polígono tracejado.

Uma versão mais restrita do *crossover* de um ponto fora proposta por Poli e Langdon (1997) um ano antes. Seu funcionamento é similar ao *crossover* de um ponto, diferindo apenas pelo fato de que os pontos escolhidos devem representar funções iguais.

2.7.2.3 *Size fair crossover*

No *size fair crossover*, a escolha do ponto de cruzamento da primeira árvore T_1 ocorre de maneira semelhante ao *crossover* de um ponto, ou seja, uma escolha aleatória em que 90% das vezes ocorre nos nós internos (funções) e os outros 10% nos nós folha (terminais). A sub-árvore do ponto selecionado de T_1 é apagada para dar lugar à sub-árvore selecionada na segunda árvore T_2 .

A diferença reside em uma restrição na escolha do ponto de T_2 : se N_{t_1} é o tamanho da sub-árvore deletada de T_1 , então o nó selecionado em T_2 deve ser escolhido entre os que são raízes de sub-árvores de tamanho máximo $1 + 2N_{t_1}$ (LANGDON, 2000).

A restrição de tamanho imposta à escolha do segundo ponto contribui para evitar o “inchaço” da árvore – o efeito *bloat*. Segundo Langdon (2000), sem esta restrição o tamanho médio da população após 50 gerações aumenta cerca de 2,5 vezes.

2.7.2.4 *Crossover* uniforme

A analogia com GA levou a outra vertente: o *crossover* uniforme. Nesta, semelhante ao que ocorre na versão de um ponto, o conjunto aceitável para se trabalhar localiza-se na região comum entre as árvores. Porém, um subconjunto de nós é escolhido para ser permutado entre os indivíduos. O processo é descrito nas Figuras 14 e 15.

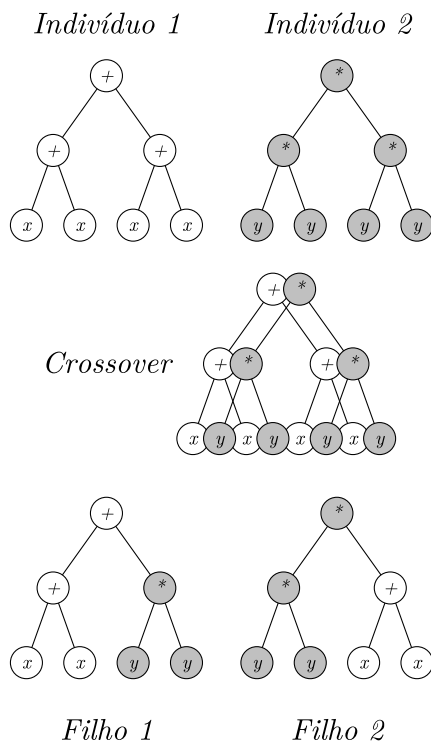


Figura 12 – *Crossover* de um ponto em duas árvores de mesmo formato. Qualquer nó da árvore pode ser escolhido como ponto de recombinação. Adaptado de Poli e Langdon (1998).

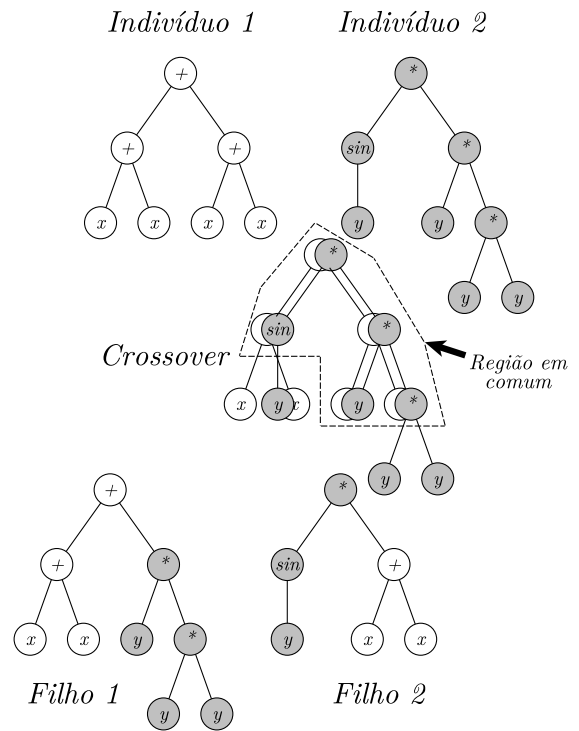


Figura 13 – *Crossover* de um ponto em duas árvores de formatos diferentes. Qualquer nó da área em comum pode ser escolhido como ponto de recombinação. Adaptado de Poli e Langdon (1998).

2.7.2.5 *Crossover* de preservação de contexto

D’haeseleer (1994) propôs um operador de cruzamento que tenta preservar o contexto das sub-árvores que aparecem nos pais. Para utilizá-lo, cada nó da árvore é identificado de forma única por meio de uma definição de localização. Como pode ser visto na Figura 16, a cada nó é atribuído uma tulpá $T = (b_1, \dots, b_i, \dots, b_n)$, em que n é a profundidade do nó e b_i indica qual aresta foi escolhida no nível i , contando da esquerda para a direita.

Este operador pode ser implementado de duas formas: a “forte” ou *Strong Context Preserving Crossover* (SCPC) e a “fraca” ou *Weak Context Preserving Crossover* (WCPC). No SCPC, apresentado na Figura 17, o cruzamento ocorre somente nos pontos cujas coordenadas são exatamente iguais. Essa restrição pode causar problemas na diversidade da população, uma vez que os nós de uma região da árvore dificilmente serão redistribuídos para outras regiões. O operador de mutação pode ser utilizado para contornar esse problema.

WCPC é menos restrito. Seja T_1 e T_2 os conjuntos de nós em comum dentro das árvores 1 e 2. A seleção do nó da primeira árvore é feita da mesma maneira que no SCPC, ou seja, escolhendo um nó $T'_1 \in T_1$. A diferença reside na escolha do nó T'_2 na

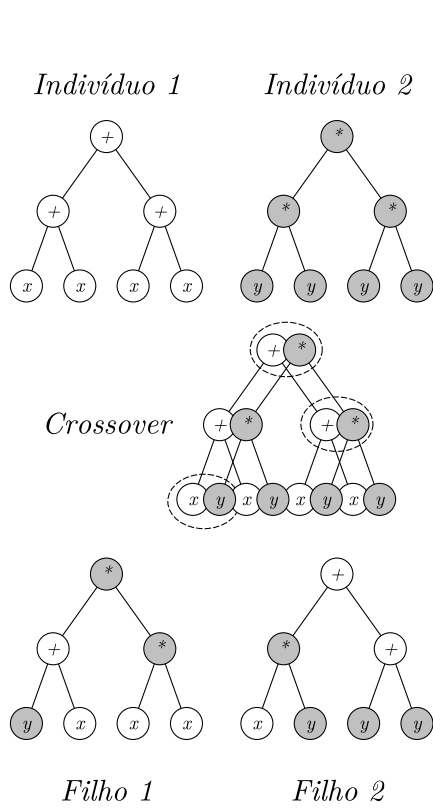


Figura 14 – *Crossover* uniforme em duas árvores de mesmo formato. Um subconjunto randomicamente selecionado do conjunto de pontos formador é permutado entre as árvores. Adaptado de Poli e Langdon (1998).

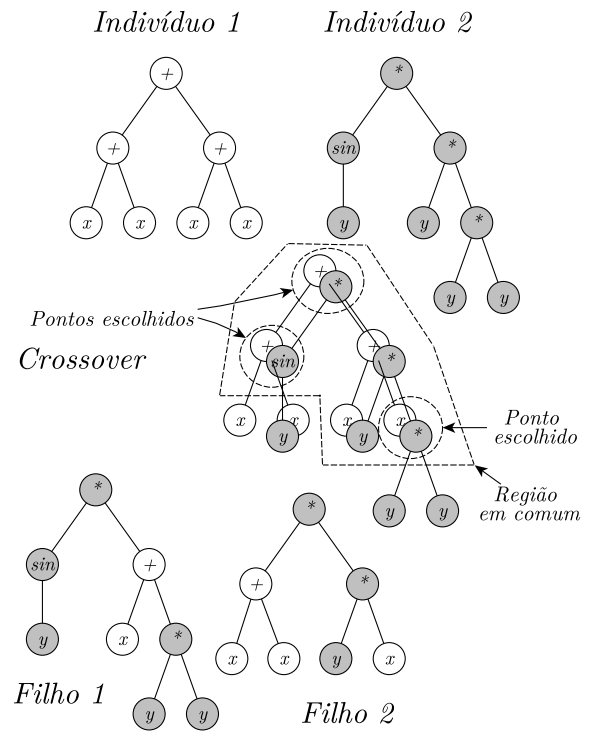


Figura 15 – *Crossover* uniforme em duas árvores de formatos diferentes. Um subconjunto randomicamente selecionado do conjunto de pontos da região em comum é permutado entre as árvores. Adaptado de Poli e Langdon (1998).

segunda árvore: este é randomicamente selecionado dentro de T_2 .

D'haeseleer (1994) aplicou esses dois cromossomos em problemas clássicos de programação genética encontrados em Koza (1994) e Koza (1992b), apresentando, em muitos casos, performance superior às versões com o *crossover* regular.

2.7.3 Mutação

A mutação é um operador que executa alterações estruturais aleatórias na população, reintroduzindo, dessa maneira, a diversidade populacional, o que evita a convergência prematura. É um operador assexuado: a partir de um único indivíduo, produz um único filho (KOZA, 1992b).

Em Koza (1992b), a mutação é classificada como um operador secundário, sob o argumento de que a GP não é uma simples busca aleatória, não necessitando, assim, desse operador. Koza (1992b) apresenta resultados de experimentos de evolução de funções booleanas que demonstram a pouca influência advinda do uso da mutação. Há, no entanto,

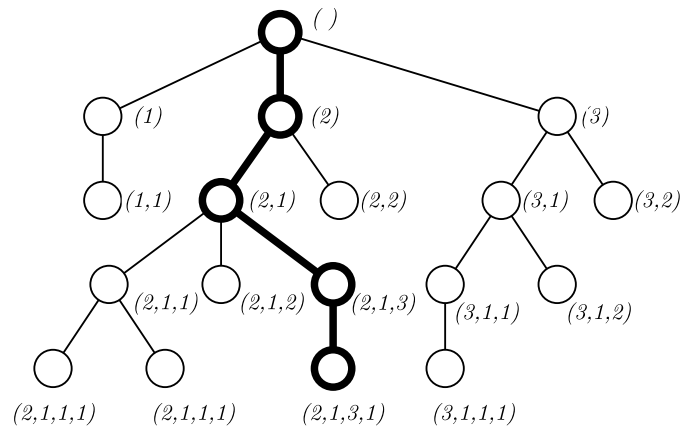


Figura 16 – Coordenadas marcadas em uma árvore para *crossover* de preservação de contexto. O nó (2, 1, 3, 1) foi alcançado passando pela segunda aresta da raiz, seguindo pela primeira aresta, seguida da terceira e da primeira. Adaptado de D’haeseleer (1994).

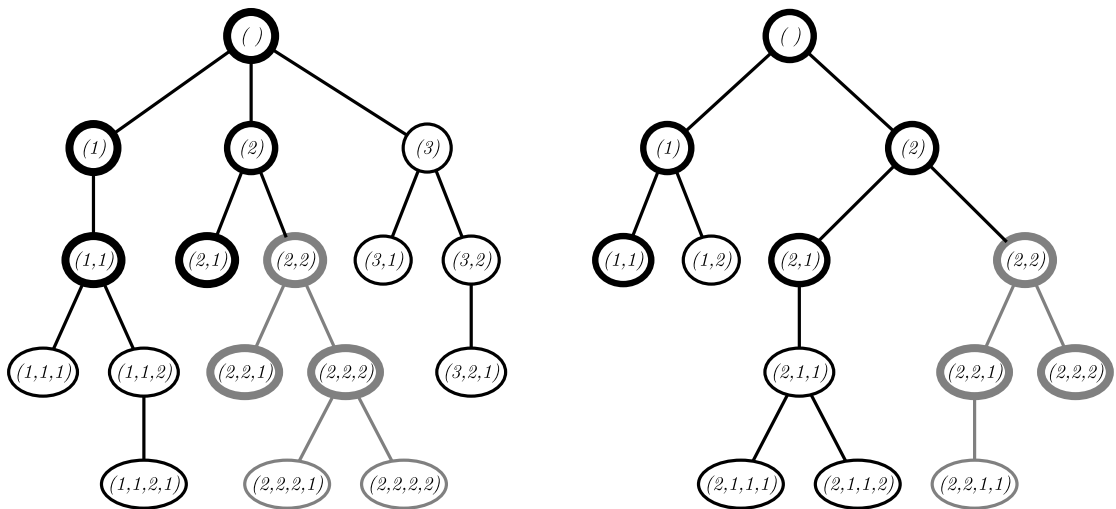


Figura 17 – Lógica de escolha de pontos no *crossover* de preservação de contexto. Todos os nós com linhas mais espessas podem ser utilizados como ponto de cruzamento. Em cinza, tem-se as subárvores que podem ser permutadas. Adaptado de D’haeseleer (1994).

pesquisadores como O’Reilly e Oppacher (1994), Chellapilla (1997), Harries e Smith (1997) e Luke e Spector (1997) que defendem sua utilização, alegando que a mutação, principalmente quando combinada com outros operadores, pode trazer benefícios para problemas específicos de GP. Por essa razão, optou-se por utilizar a mutação no presente trabalho.

2.7.3.1 Tipos de mutações

A primeira versão do operador de mutação voltada para GP foi proposta por Koza (1992b) e é conhecida por *mutação de sub-árvore* – ou *subtree mutation*. Consiste em selecionar randomicamente um nó da árvore e substituí-lo por uma sub-árvore criada também de forma aleatória. Uma versão semelhante foi proposta por Kinneer (1994), que impôs uma restrição: o tamanho da nova árvore gerada não poderia ser 15% maior do que a original.

Outra abordagem que também considera o controle do tamanho da árvore resultante (prevenindo, assim, o efeito *bloat*) é o *size-fair subtree mutation* (LANGDON, 1998). Nesta versão, a nova sub-árvore é substituída por outra cujo tamanho é um valor no intervalo $[l/2, 3l/2]$, onde l é o tamanho da sub-árvore a ser substituída.

McKay, Willis e Barton (1995) propuseram o operador *point mutation* que, em analogia com o que é feito em GA, substitui um nó qualquer da árvore por outro, aleatoriamente criado. Esta ação sempre cria descendentes de mesmo tamanho que seus pais. Para o caso de nós função, acrescenta-se a restrição de trocá-lo por outro com a mesma quantidade de argumentos, o que garante a integridade da árvore (MCKAY; WILLIS; BARTON, 1995). Uma versão mais restrita desse operador é o *shrink mutation*, proposto por Angeline (1996), que escolhe um ponto qualquer da árvore (seja ele terminal ou função) e o substitui por um terminal criado aleatoriamente, gerando, predominantemente, descendentes menores.

O operador *hoist mutation*, proposto por Kinneer (1994), tem a mesma preocupação de não gerar descendentes maiores que seus pais. Ele cria novas árvores a partir de alguma sub-árvore da árvore pai, escolhida aleatoriamente.

2.7.4 Edição

Assim como a mutação e a permutação, o operador de edição é assexuado que age sob um único indivíduo por vez, gerando um único filho. Este operador permite editar e simplificar a expressão gerada pela GP, aplicando recursivamente conjuntos de regras gerais e específicas. As regras gerais aplicam-se quando uma função sem efeitos colaterais e independente do contexto tem como argumentos apenas constantes. Neste caso, o valor retornado é calculado e este substitui o nó função. Regras pré-estabelecidas como simplificações aritméticas ou lógicas também se enquadram nessa categoria (KOZA, 1992b).

Koza (1992b) utiliza este operador por duas razões: para simplificar a saída, tornando o resultado mais “legível” e para utilizá-lo durante a execução, simplificando as expressões de maneira a reduzir a carga total de processamento. Seus experimentos não foram conclusivos quanto ao acréscimo de qualidade na resposta encontrada.

Neste trabalho, operador de edição foi utilizado sob outra perspectiva: tornar o resul-

tado da execução da GP, sempre que possível, livre das operações protegidas, trabalhando simbolicamente. Por exemplo, sempre que uma expressão do tipo $y/(\cos(x-1) - (\cos(x-1)))$ for encontrada, todo este trecho é substituído por 1 na resposta simplificada. Este procedimento garante que a equação encontrada esteja sintaticamente correta, podendo servir de entrada para outros programas como o Gnuplot¹ ou o Maxima².

2.7.5 Permutação

Este é um operador assexuado que age sob um único indivíduo por vez e que gera um único filho. A seleção do indivíduo ocorre da mesma forma que é feita para os operadores de reprodução e de recombinação.

Tendo selecionado o indivíduo, o operador age selecionando randomicamente um nó função dentro da árvore. Caso essa função possua k argumentos, suas ordens são trocadas por uma das $k!$ permutações possíveis, que é também escolhida ao acaso.

Koza (1992b) não observou, em média, muitos benefícios deste operador sob a aptidão do indivíduo. Maxwell (1996), no entanto, obteve mais sucesso com uma variação deste operador, chamada de *swap*, que executa permutações somente em funções binárias não comutativas.

2.8 A escolha de parâmetros

Não existem fórmulas definidas para a escolha dos parâmetros de controle em programação genética. Há apenas um conjunto de valores que se convencionou usar, frequentemente utilizado na literatura. Esses valores foram obtidos empiricamente por uma grande quantidade de pesquisadores.

Pesquisadores como Poli, Langdon e McPhee (2008), Koza (1992a) e Araújo (2004) apontam o tamanho da população e a quantidade de gerações como os parâmetros mais importantes, recebendo os valores 500 e 51, respectivamente. A quantidade de gerações foi determinado empiricamente por diversos pesquisadores na primeira metade da década de 1990, sob o argumento de que poucas diferenças ocorriam após a geração 51.

Em programação genética, convencionou-se o uso excludente dos operadores genéticos: se a recombinação é aplicada sob determinado indivíduo, este não será submetido à reprodução e/ou à mutação, por exemplo. A taxa de cruzamento costuma ser elevada, com valores por volta de 85%. A mutação, conforme discutido na Seção 2.7.3, é alvo de controvérsias e, no presente trabalho, foi utilizada com taxa de 5%. Os outros 10% costumam ser empregados na reprodução.

¹ Software livre para manipulação de gráficos. Maiores informações em: <http://www.gnuplot.info/>.

² Software utilizado na manipulação algébrica simbólica. No contexto de regressão simbólica via GP, pode ser útil para simplificar equações de maneira mais precisa.

Os operadores de edição e permutação raramente são usados e seus efeitos, conforme discutido neste capítulo, não apresentam acréscimos significativos de qualidade. Por essa razão, neste trabalho a edição é utilizada apenas para simplificar a resposta final dada ao programa e a permutação não é usada.

Convenciona-se também as profundidades mínima e máxima das árvores geradas via aplicação dos operadores genéticos: 6 e 17, respectivamente.

Redução do efeito *bloat* via dominância de Pareto

A programação genética permite a representação de genótipos de tamanho e forma arbitrários. Esta é uma das principais diferenciações entre a GP e as demais abordagens existentes no campo da computação evolutiva. Tal “liberdade de representação” é acompanhada de um problema observado desde as primeiras implementações realizadas por Koza (1992a) e que pode ser visto na Figura 18: o crescimento descontrolado do tamanho médio das árvores que compõem a população, sem que haja a consequente melhoria de sua aptidão média (POLI; LANGDON; MCPHEE, 2008). Este “inchaço” das árvores é conhecido como efeito *bloat* e é caracterizado pelo aparecimento dos chamados *introns* nas árvores – trechos (ou sub-árvores) que não apresentam qualquer efeito na avaliação do indivíduo. Embora o *bloat* seja também observado em outras áreas (redes neurais, autômatos, etc.), é na programação genética que ele causa os maiores problemas (LUKE; PANAIT, 2006).

As raízes teóricas do efeito *bloat* ainda não estão totalmente elucidadas. Há, por exemplo, em Langdon et al. (1999) grande discussão sobre esse assunto, com diversas propostas de operadores genéticos que diminuam este efeito, mas suas causas não são abordadas. Mesmo não havendo clareza quanto à sua dinâmica, grandes esforços são empreendidos no sentido de diminuir o *bloat*. As principais razões para tanto, em acordo com Luke e Panait (2006), são:

- ❑ É preferível que uma solução seja a mais simples e compacta possível;
- ❑ O sistema deve naturalmente direcionar a busca para regiões no espaço onde se encontram as árvores mais compactas e de melhores aptidões;
- ❑ Quanto maiores são as árvores, mais esforço computacional é necessário para o processo de avaliação e mais memória física é requerida para acomodá-las. O *bloat* pode causar o consumo total da memória disponível, comprometendo todo o sistema.

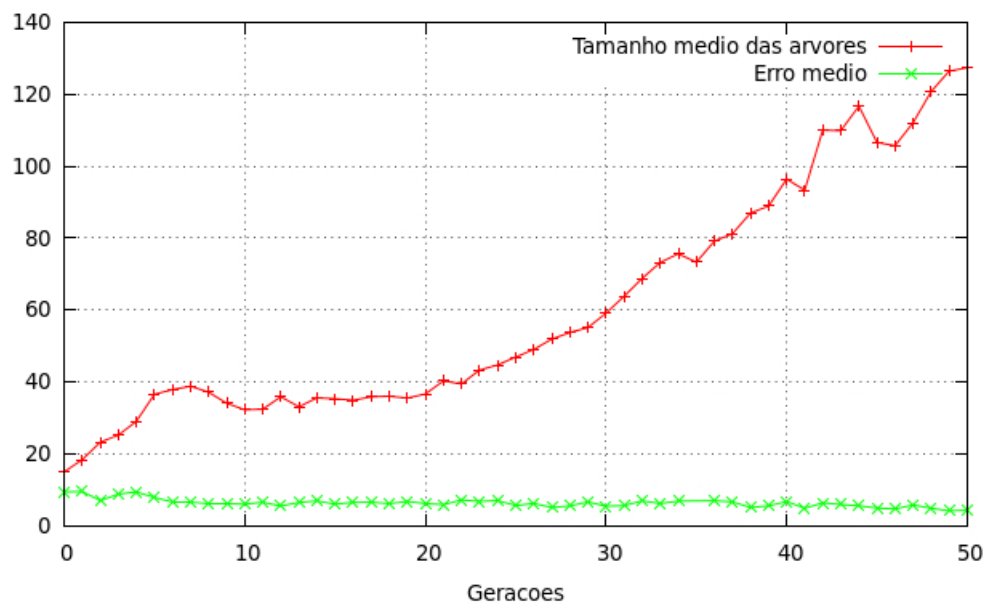


Figura 18 – Crescimento do tamanho médio das árvores que compõem uma população no processo de regressão simbólica da função $x^4 + x^3 + x^2 + x$ no intervalo $[-1, 1]$. Percebe-se que a partir da geração 20, o tamanho médio das árvores da população cresce de forma praticamente exponencial, enquanto o valor médio do erro da população permanece praticamente inalterado.

O método mais comum de combate ao *bloat* é o proposto por Koza (1992a) e consiste em limitar o tamanho máximo das árvores geradas. Nessa abordagem, sempre que uma árvore de profundidade maior que 17 é gerada, esta é ignorada e seus pais são copiados para a próxima geração. Pequenas variações deste método são encontradas em Luke (2000a), Luke (2003).

Este capítulo utiliza uma vertente da programação genética, conhecida como programação genética com Pareto, que faz o balanceamento entre dois objetivos (minimização do erro e minimização do tamanho da árvore) para combater o efeito *bloat*.

3.1 Algoritmos evolutivos multiobjetivos

Os algoritmos evolutivos multiobjetivos são heurísticas comumente utilizadas no tratamento de problemas de otimização cujo conjunto de soluções viáveis é composto por elementos que atendam a dois ou mais objetivos. Ao se trabalhar com problemas multiobjetivos, percebe-se a impossibilidade de otimizar todos os objetivos de forma simultânea, visto que o atendimento a uma restrição específica pode levar ao comprometimento das demais. De fato, é comum que haja problemas cujas melhores soluções não são capazes de atender a todos os requisitos ao mesmo tempo. Estes conceitos foram aplicados aos algoritmos genéticos.

As pesquisas em algoritmos genéticos multiobjetivos ocorrem desde a década de 1980. Sua primeira implementação, feita por Schaffer (1985), fazia a avaliação de cada indivíduo separadamente. Percebe-se que ao tratar problemas com m objetivos, um caminho natural é buscar uma função de aptidão f que agregue todos os objetivos simultaneamente, por meio da atribuição de pesos percentuais w em cada objetivo. Esta função seria da forma

$$f(x) = \sum_{i=1}^m w_i f_i(x), \text{ com } \sum_{i=1}^m w_i = 1. \quad (11)$$

Quando o vetor de pesos w existe, sua determinação pode não ser tarefa simples. Uma alternativa é tratar cada objetivo m separadamente e aplicar algum tipo de análise para determinar se uma solução a é melhor que outra solução b . Em casos como este, o conceito de *dominância de Pareto* pode ser útil: diz-se que a domina b (o que é representado por $a \prec b$) se, e somente se:

- a não é pior que b em qualquer objetivo; e
- a é estritamente melhor que b em pelo menos um objetivo.

A dominância é uma relação de ordem parcial e obedece apenas às propriedades irreflexiva ($a \prec b \Rightarrow a \neq b$), assimétrica ($a \prec b \Rightarrow \neg(b \prec a)$) e transitiva ($a \prec b \wedge b \prec c \Rightarrow a \prec c$). Isso significa que podem existir soluções a e b tais que a não domine b e nem b domine a , formando subconjunto de soluções não dominadas. O subconjunto que possui elementos que não são dominados por quaisquer outro elemento é chamado fronteira ótima de Pareto, como pode ser visto na Figura 19.

Goldberg (1989) foi o primeiro a usar o conceito de dominância de Pareto para tratar problemas multiobjetivos via algoritmos genéticos, atrelando a aptidão de um indivíduo à quantidade de soluções que ele domina e ao espalhamento do indivíduo junto à fronteira de Pareto. Fonseca e Fleming (1995) expandiram este conceito, classificando a população em diferentes níveis, com cada nível contendo elementos que dominam igual quantidade de elementos. Nesta abordagem, a aptidão de um indivíduo é a média das aptidões dos elementos do nível a que ele pertence.

A classificação da população por níveis de dominância foi também a abordagem utilizada por Srinivas e Deb (1994) no *Non-dominated Sorting Genetic Algorithm* (NSGA). Este algoritmo diferencia-se dos anteriores por utilizar o conceito de “distância de multidão”, medida que indica o quão distante uma solução encontra-se dos seus vizinhos – soluções anteriores e posteriores dentro da mesma fronteira de dominância. Durante o processo de seleção, é utilizado o torneio de forma que as soluções pertencentes aos primeiros níveis são preferíveis e, dadas duas soluções no mesmo nível, prefere-se aquela que esteja mais isolada, a fim de melhor explorar a região do espaço de busca onde ela se encontra. Apesar de apresentar conceitos interessantes do ponto de vista teórico, o NSGA foi alvo de diversas críticas, principalmente por apresentar alto custo computacional. Além disso,

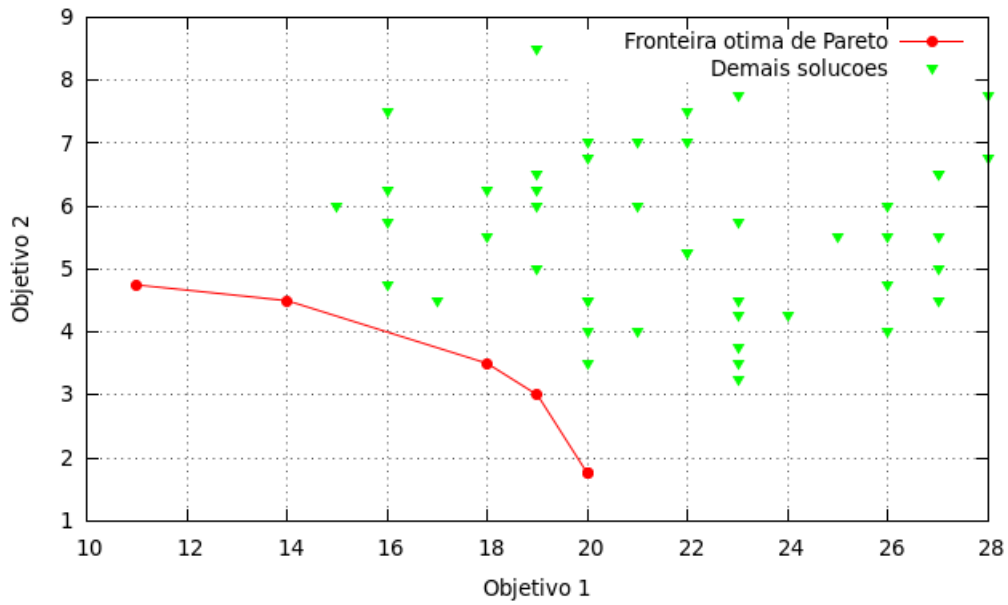


Figura 19 – Conjunto de soluções que tentam minimizar dois objetivos. Os pontos representados por $\bullet$ correspondem às soluções não dominadas e formam a fronteira ótima. É deste subconjunto que se extrai a solução mais viável. Os demais pontos são representados por $\blacktriangledown$.

ele requer que seja definida uma constante σ_{share} (parâmetro de compartilhamento), valor que deve ser empiricamente definido de acordo com a aplicação.

Zitzler e Thiele (1999) propuseram o *Strength Pareto Evolutionary Algorithm* (SPEA), método que, além de utilizar os conceitos de dominância, mantém um arquivo separado da população corrente com as soluções não dominadas encontradas até então. Para o cálculo da aptidão de um indivíduo, considera-se quantos indivíduos são dominados e quantos o dominam. A segunda versão do SPEA, o *Strength Pareto Evolutionary Algorithm 2* (SPEA2) foi lançada anos depois (ZITZLER; LAUMANN; THIELE, 2001) para prover melhorias no que se refere à atribuição de aptidão e ao processo de seleção do indivíduo, dentre outros.

O SPEA2 foi utilizado por Bleuler et al. (2001) para controlar o efeito *bloat* em programação genética estabelecendo dois objetivos: 1) encontrar soluções que sejam compactas; 2) com o erro próximo (ou igual) a zero. Neste trabalho, o SPEA2 apresentou bons resultados quando aplicado na geração de equações booleanas para problemas de paridade par de diversos tamanhos de entrada.

Vladislavleva (2008) realizou estudos abrangentes sobre programação genética com Pareto para modelagem via regressão simbólica, no qual foi mostrado a superioridade desta técnica quando comparada à programação genética padrão. Seus testes foram realizados em uma *Toolbox* do software *Matlab* chamada *ParetoGP*, que internamente utiliza a segunda versão do algoritmo NSGA (DEB et al., 2002). Como um dos objetivos deste

trabalho é desenvolver uma ferramenta para programação genética, estudos mais detalhados deste algoritmo e seu uso na programação genética com Pareto serão apresentados nas próximas seções.

3.2 NSGA-II

Embora o algoritmo NSGA, proposto em Srinivas e Deb (1994), destaque-se por ter sido a primeira abordagem em algoritmos genéticos multiobjetivos a explorar as múltiplas fronteiras de dominância, ele foi largamente criticado, tendo em vista três pontos principais (DEB et al., 2002):

- ❑ Alta complexidade computacional exigida na determinação das fronteiras de dominância: $O(MN^3)$, com M sendo a quantidade de objetivos e N , o tamanho da população;
- ❑ Ausência de elitismo, que, segundo Rudolph (1999) e Zitzler, Deb e Thiele (2000), aumentam o desempenho do AG e previne as perdas de boas soluções;
- ❑ A necessidade de especificação do parâmetro de compartilhamento σ_{share} , uma constante que auxilia na garantia de diversidade, mas cujo valor deve ser empiricamente determinado.

Em Deb et al. (2002), foi proposta nova versão do NSGA, o NSGA-II, que solucionou as três questões destacadas acima e mostrou, por meio de testes e análises, a superioridade desta segunda versão à primeira e também a outras técnicas evolutivas que utilizam as fronteiras de Pareto. O NSGA-II é um algoritmo executado em três fases distintas: o *fast non-dominated sort*, que distribui o conjunto de soluções em suas fronteiras correspondentes, o *crowding-distance*, que evita a convergência prematura por meio da manutenção de diversidade e a maneira como estes dois conceitos são aplicados na evolução de uma população. Cada fase é detalhada a seguir.

3.2.1 Fast non-dominated sort

No *fast non-dominated sort*, para cada solução p , dentro de um conjunto de soluções P de tamanho N , são determinados:

1. n_p : o número de soluções em P que dominam p , e
2. S_p : o conjunto de soluções dentro de P que são dominadas por p .

Inicialmente, o conjunto P é percorrido e cada solução $p \in P$ é comparada com as demais, a fim de determinar a quantidade de soluções que dominam p e identificar os elementos dominados por p , que serão armazenados em S_p . O processo de determinação

de n_p e S_p , para todo $p \in P$, demanda tempo de execução $O(MN^2)$. Ao final desse processamento, as soluções p que apresentarem $n_p = 0$ (não são dominadas por nenhuma outra) são identificadas como pertencentes à primeira fronteira e colocadas em $\mathcal{F}_1$. A seguir, os integrantes q de S_p têm os valores de n_q decrementados em uma unidade. Análises semelhantes são feitas para todos os $q \in S_p$, a fim de identificar as demais fronteiras. A lógica de funcionamento do *Fast non-dominated sort* é melhor explicada no Algoritmo 5.

Algoritmo 5 Procedimento *fast-non-dominated-sort*(P)

```

1: para todo  $p \in P$  faça
2:    $S_p \leftarrow \emptyset$ 
3:    $n_p \leftarrow 0$ 
4:   para todo  $q \in P$  faça
5:     se  $p \prec q$  então
6:        $S_p \leftarrow S_p \cup \{q\}$   (se  $q$  domina  $p$ ,  $q$  é adicionado ao conjunto de soluções que dominam  $p$ )
7:     senão se  $q \prec p$  então
8:        $n_p \leftarrow n_p + 1$   (caso contrário, incrementa o contador que diz quantas soluções dominam  $p$ )
9:     fim se
10:  fim para
11:  se  $n_p = 0$  então
12:     $p_{rank} \leftarrow 1$ 
13:     $\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cup \{p\}$   (se  $p$  não é dominado por nenhuma solução, então ele pertence à primeira fronteira)
14:  fim se
15: fim para
16:  $i \leftarrow 1$   (inicializa o contador de fronteiras)
17: enquanto  $\mathcal{F}_i \neq \emptyset$  faça
18:    $Q \leftarrow \emptyset$   (usado para armazenar os membros da próxima fronteira)
19:   para todo  $p \in \mathcal{F}_i$  faça
20:     para todo  $q \in S_p$  faça
21:        $n_q \leftarrow n_q - 1$ 
22:       se  $n_q = 0$  então
23:          $q_{rank} \leftarrow i + 1$ 
24:          $Q \leftarrow Q \cup \{q\}$   (entre as soluções restantes,  $q$  não é dominado por nenhuma, fazendo parte, então, da próxima fronteira.)
25:       fim se
26:     fim para
27:   fim para
28:    $i \leftarrow i + 1$ 
29:    $\mathcal{F}_i \leftarrow Q$ 
30: fim enquanto

```

A fronteira que uma solução pertence é chamada também de *rank* da solução. Dessa forma, as soluções pertencentes à fronteira $\mathcal{F}_1$ (Pareto-ótimo) possuem $rank = 1$, e assim sucessivamente.

3.2.2 Crowding-distance

Para evitar que a população convirja para ótimos locais, há a necessidade de se estabelecer critérios de preservação da diversidade populacional ao longo da evolução, para que esta possa, da melhor maneira possível, estar distribuída em todo (hiper)espaço de busca. O NSGA usa como métrica o parâmetro de compartilhamento σ_{share} , uma constante definida pelo usuário que serve para calcular a distância média entre duas soluções pertencentes a uma mesma fronteira – a escolha das soluções mais isoladas é preferível, pois a aplicação dos operadores genéticos pode levar a regiões ainda não exploradas do espaço de busca. Duas são as complicações inerentes à utilização do σ_{share} apontadas por Deb et al. (2002): 1) a efetiva utilidade deste método é dependente da escolha adequada de σ_{share} , ou seja, o usuário é responsável por identificar as características do problema em estudo e determinar o valor desta constante, e 2) cada solução deve ser comparada a todas as outras, exigindo assim tempo de execução $O(N^2)$.

O NSGA-II dispensa o uso da constante de compartilhamento (com suas dificuldades subjacentes) por meio do operador *crowding-distance*, que consegue manter a diversidade populacional sem a necessidade de intervenção por parte do usuário e com um custo computacional menor. Dada uma solução i , em uma fronteira de Pareto, a densidade populacional é obtida calculando a distância média entre duas soluções i e as soluções vizinhas na fronteira – o que é chamado $i_{distancia}$. O *crowding-distance* é representado na Figura 20, onde se considera uma função com apenas dois objetivos cuja vizinhança (o retângulo pontilhado cercado a solução i por $i+1$ e $i-1$) pode ser facilmente representada de forma gráfica. Esta regra é válida para uma quantidade arbitrária de funções objetivo dentro do hiperespaço – o que justifica o nome “cuboide”.

O cálculo do *crowding-distance* é realizado após a execução do *fast non-dominated sort* e é aplicado a cada conjunto $\mathcal{F}_i$. Inicialmente, ao $i_{distancia}$ de cada solução é atribuído o valor zero. A seguir, as soluções da fronteira corrente são ordenadas de acordo com cada objetivo m (um por vez), para estabelecer o conceito de vizinhança necessário ao algoritmo. À primeira solução e à última, são atribuídos um valor infinito. Para as demais soluções, o valor de $i_{distancia}$ será igual à diferença absoluta entre as distâncias das soluções adjacentes (ou seja, $(i-1)_{distancia}$ e $(i+1)_{distancia}$). Este cálculo deve ser normalizado, por isso, a diferença é dividida pela diferença entre os valores máximo e mínimo ($f_m^{max} - f_m^{min}$) da solução para o objetivo corrente.

A sequência completa da execução do *crowding-distance* pode ser vista em detalhes no Algoritmo 6, que recebe uma fronteira $\mathcal{I}$. É importante notar que na notação do algoritmo, $\mathcal{I}[i].m$ significa “o valor da solução i para o objetivo m ”. Como o algoritmo de ordenação mais eficiente que se conhece possui tempo de execução $O(n \log n)$ (CORMEN et al., 2001), o conjunto P possui N elementos e a quantidade de objetivos é M , conclui-se que o algoritmo possui complexidade $O(MN \log N)$.

Quando todas as soluções estiverem com seus *rank* e *crowding-distance* determinados, é

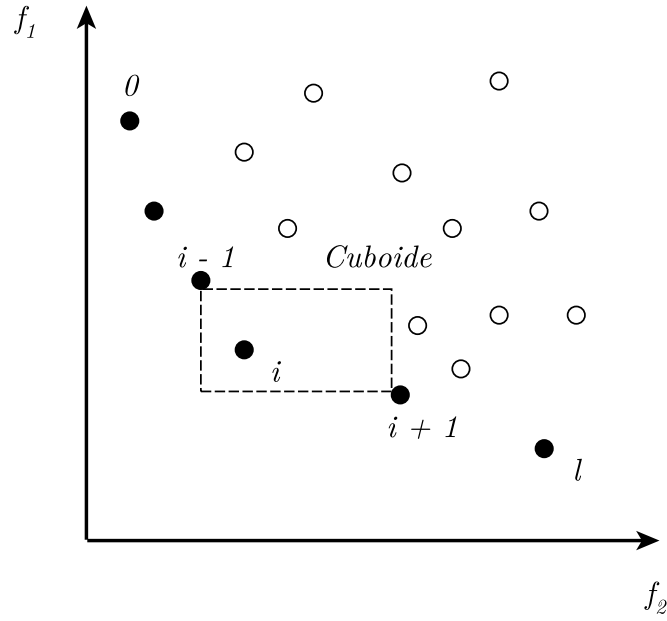


Figura 20 – Cálculo do *crowding-distance*, adaptado de Deb et al. (2002). Os círculos cheios representam as soluções pertencentes ao Pareto-ótimo.

Algoritmo 6 Procedimento *crowding-distance-assignment*($\mathcal{I}$)

```

1:  $l \leftarrow |\mathcal{I}|$    (número de soluções em  $\mathcal{I}$ )
2: para todo  $i$  faça
3:    $\mathcal{I}[i]_{distancia} = 0$    (inicializa as distâncias)
4: fim para
5: para todo objetivo  $m$  faça
6:    $\mathcal{I} \leftarrow \text{ordena}(\mathcal{I}, m)$    (ordena  $\mathcal{I}$ , em ordem crescente, de acordo com o objetivo  $m$ )
7:    $\mathcal{I}[1]_{distancia} \leftarrow \mathcal{I}[l]_{distancia} \leftarrow \infty$    (os pontos das fronteiras sempre serão selecionados)
8:   para  $i = 2$  até  $l - 1$  faça
9:      $\mathcal{I}[i]_{distancia} \leftarrow \mathcal{I}[i]_{distancia} + (\mathcal{I}[i + 1].m - \mathcal{I}[i - 1].m) / (f_m^{max} - f_m^{min})$ 
10:  fim para
11: fim para

```

possível aplicar o operador de seleção *crowded-comparison*, um operador de ordem parcial denotado por $\prec_n$. Dadas duas soluções a e b , $a \prec_n b$ apenas se uma das condições se verificar:

1. $a_{rank} < b_{rank}$
2. $(a_{rank} = b_{rank}) \wedge (a_{distancia} > b_{distancia})$.

Com este operador, é dada preferência pelas soluções que se encontram nas primeiras fronteiras. Quando elementos de mesma fronteira são comparados, as soluções mais isoladas são escolhidas.

3.2.3 O *loop* principal

O algoritmo se inicia com a criação de uma população inicial aleatória P_0 , de tamanho N . Através do *fast non-dominated sort*, o *rank* (índice da fronteira) de cada elemento de P_0 é determinado. A seguir, aplica-se um torneio binário baseado em *rank* para selecionar elementos de P_0 que serão submetidos aos operadores de cruzamento e mutação. Os filhos gerados são usados para povoar uma nova população Q_0 , também de tamanho N . Esta etapa ocorre antes da primeira geração. Quando a evolução inicia, o conjunto P_t é usado para armazenar as melhores soluções encontradas até a geração t (elitismo) e Q_t abriga as soluções resultantes da aplicação dos operadores genéticos em P_t .

A cada geração t , as populações P_t e Q_t são mescladas no mesmo conjunto R_t , de tamanho $2N$, e seus elementos são separados de acordo com suas fronteiras de dominância $\mathcal{F}_i$. Se o tamanho de $\mathcal{F}_1$, que contém as melhores soluções, for menor ou igual a N , então todo este conjunto é copiado para P_{t+1} . Caso contrário, P_{t+1} é completado com os elementos das fronteiras subsequentes. A seguir, os valores de $i_{distancia}$ são calculados para todos os elementos i de P_{t+1} e estes são selecionados, por meio de torneios binários com o operador $\prec_n$, para serem submetidos aos operadores genéticos e gerarem descendentes para povoar a próxima população Q_{t+1} . O procedimento completo é detalhado no Algoritmo 7 e a representação gráfica do ciclo principal, em suas duas fases, é dada na Figura 21.

Algoritmo 7 Algoritmo genético com NSGA-II

- 1: $P_0 \leftarrow \text{cria-populacao-aleatoria}(N)$ (*cria população aleatória de tamanho N*)
 - 2: $Q_0 \leftarrow \text{criar-nova-populacao}(P_0)$ (*use seleção, crossover e mutação para criar a população Q_1*)
 - 3: $t \leftarrow 1$ (*contador de gerações*)
 - 4: **enquanto** $t \leq \text{GERACOES}$ **faça**
 - 5: $R_t \leftarrow P_t \cup Q_t$ (*une as duas populações*)
 - 6: $\mathcal{F} \leftarrow \text{fast-non-dominated-sort}(R_t)$ ($\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \dots)$, *separa as fronteiras de não-dominância de R_t*)
 - 7: $P_{t+1} \leftarrow \emptyset$
 - 8: $i \leftarrow 1$
 - 9: **enquanto** $|P_{t+1}| + |\mathcal{F}_i| \leq N$ **faça**
 - 10: $\text{crowding-distance-assignment}(\mathcal{F}_i)$ (*determina os crowding-distance de $\mathcal{F}_i$*)
 - 11: $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i$ (*inclui a i -ésima fronteira de não-dominância na população de pais*)
 - 12: $i \leftarrow i + 1$ (*marca a próxima fronteira para a inclusão*)
 - 13: **fim enquanto**
 - 14: $\text{ordena}(\mathcal{F}_i, \prec_n)$ (*ordena em ordem decrescente usando $\prec_n$*)
 - 15: $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i[1 : (N - |P_{t+1}|)]$ (*escolhe os primeiros $(N - |P_{t+1}|)$ elementos*)
 - 16: $Q_{t+1} \leftarrow \text{criar-nova-populacao}(P_{t+1})$ (*use seleção, crossover e mutação para criar a nova população Q_{t+1}*)
 - 17: $t \leftarrow t + 1$ (*incrementa o contador de gerações*)
 - 18: **fim enquanto**
-

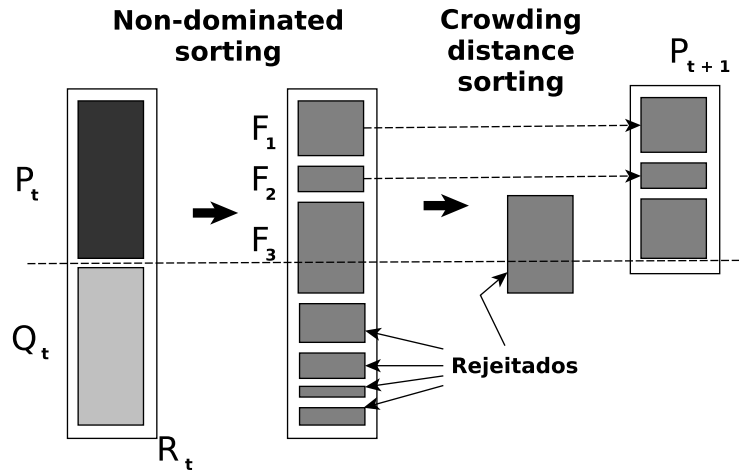


Figura 21 – Representação do procedimento do NSGA-II. A junção dos elementos de P_t e Q_t e sua correta separação nas fronteiras de dominância auxiliam na preservação dos melhores indivíduos – elitismo.

3.3 Programação genética com Pareto

O tratamento do efeito *bloat* exige que, durante a evolução da população, sejam considerados aptos os indivíduos que melhor atendam a dois objetivos principais: menor erro e menor complexidade estrutural. Em Bleuler et al. (2001), é proposta uma abordagem multiobjetivo para tratar esse problema por meio do algoritmo SPEA2 (ZITZLER; LAUMANN; THIELE, 2001). Vladislavleva (2008), baseando-se principalmente nos resultados de Smits e Kotanchek (2004), mostra em sua tese a eficácia e a eficiência da programação genética com Pareto para a geração de equações precisas e compactas, utilizadas na modelagem via regressão simbólica.

A programação genética com Pareto diferencia-se da abordagem clássica por dois motivos principais:

1. Por utilizar os conceitos de dominância de Pareto, vistas nas seções anteriores, para atender aos objetivos de minimização do erro e minimização da complexidade estrutural;
2. Por manter um “arquivo” com os melhores indivíduos encontrados até a geração atual, de maneira que estes participem ativamente na evolução das gerações seguintes – ou seja, proporcionando elitismo.

Os caminhos para atender o primeiro objetivo (menor erro) foram apresentadas na Seção 2.5. Já para o segundo objetivo, Smits e Kotanchek (2004), ao considerar a GP baseada em árvores, elencam várias opções:

- **Profundidade da árvore** – a quantidade de níveis existentes;

- ❑ **Número de nós** – quantidade de terminais e não-terminais existentes;
- ❑ **Número de variáveis** – pode ser considerado tanto a quantidade total que aparece nos nós-folha quanto a quantidade de variáveis distintas que aparece na equação;
- ❑ **Combinações** das opções anteriores.

A melhor métrica para complexidade é ainda um assunto em aberto. Nos trabalhos de Smits e Kotanchek (2004) e Vladislavleva (2008), por exemplo, é utilizada a minimização da soma da quantidade de nós presentes em cada sub-árvore do indivíduo. Com esta abordagem, quando duas árvores com o mesmo número de nós são comparadas, a de desenho mais simples é considerada a melhor. Entretanto, o custo computacional para a determinação deste valor cresce de maneira proporcional à quantidade de nós, razão pela qual optou-se utilizar a quantidade de nós da árvore como referência no presente trabalho.

Diferentemente do que foi feito por Vladislavleva (2008), que utilizou um conjunto de ferramentas do software *MatLab (ParetoGP Toolbox)* para obter e validar seus resultados, no presente trabalho toda a implementação de programação genética com Pareto foi realizada. Para tanto, o algoritmo NSGA-II foi adaptado ao contexto de programação genética, de forma a manter o fluxo básico do Algoritmo 7.

3.3.1 Implementação

A implementação foi realizada em linguagem Java. Versões dos algoritmos *crowding-distance-assignment* e *crowded-comparison* foram desenvolvidas de modo a utilizarem como restrições o tamanho da árvore (número de nós) e o erro apresentado (menor erro). O laço principal, que une a GP e os conceitos de dominância de Pareto podem ser analisadas no trecho de código a seguir.

```
public void executaPG() {

    ArrayArvore[] P = populacao;
    ArrayArvore[] Q = new ArrayArvore[P.length];

    for (int i = 0; i < P.length; i++) {
        avalia.apertidao(P[i]);
    }

    int k = 0;
    while (k < Q.length) {
        if (Math.random() < fcruzamento) {
            int ia = getIndividuo();
            int ib = getIndividuo();
            ArrayArvore a = P[ia].clone();
            ArrayArvore b = P[ib];
            for (int j = 0; j < avalia.getQtdSaida(); j++) {
                double[] err = P[ia].getErrosIndividuais();
```

```

        if (err[j] != 0) {
            a.crossover(j, b, tipoCrossover);
        }
    }
    Q[k++] = a;
} else if (Math.random() < fmutacao) {
    int ia = getIndividuo();
    ArrayArvore a = P[ia].clone();
    for (int j = 0; j < avalia.getQtdSaida(); j++) {
        double[] err = P[ia].getErrosIndividuais();
        if (err[j] != 0) {
            a.mutacao(j, tipoMutacao);
        }
    }
    Q[k++] = a;
} else {
    Q[k++] = P[getIndividuo()].clone();
}
}

for (int i = 0; i < Q.length; i++) {
    avalia.aptidao(Q[i]);
}

for (int g = 1; g < geracoes; g++) {

    ArrayArvore[] R = new ArrayArvore[2 * P.length];
    for (int i = 0; i < P.length; i++) {
        R[i] = P[i];
        avalia.aptidao(R[i]);
    }
    for (int i = 0; i < Q.length; i++) {
        R[i + P.length] = Q[i];
        avalia.aptidao(R[i + P.length]);
    }

    ArrayArvore[][] F = fastNonDominatedSorting(R);

    k = 0;
    int j = 0;
    while ((j < F.length) && ((k + F[j].length) < P.length))
    {
        for (int m = 0; m < F[j].length; m++) {
            populacao[k] = P[k] = F[j][m];
        }
        k++;
        j++;
    }
    if ((k < P.length) && (j < F.length)) {
        QuickSort.sortCrowding(F[j]);
        int m = 0;
        while (k < P.length) {

```

```

        populacao[k] = P[k] = F[j][m++];
    }
}
k = 0;
while (k < Q.length) {
    if (Math.random() < fcruzamento) {
        int ia = getIndividuo();
        int ib = getIndividuo();
        ArrayArvore a = P[ia].clone();
        ArrayArvore b = P[ib];
        for (j = 0; j < avalia.getQtdSaida(); j++) {
            double[] err = P[ia].getErrosIndividuais();
            if (err[j] != 0) {
                a.crossover(j, b, tipoCrossover);
            }
        }
        Q[k++] = a;
    } else if (Math.random() < fmutacao) {
        int ia = getIndividuo();
        ArrayArvore a = P[ia].clone();
        for (j = 0; j < avalia.getQtdSaida(); j++) {
            double[] err = P[ia].getErrosIndividuais();
            if (err[j] != 0) {
                a.mutacao(j, tipoMutacao);
            }
        }
        Q[k++] = a;
    } else {
        Q[k++] = P[getIndividuo()].clone();
    }
}
}
}

```

3.4 Resultados e discussões

Para testar o impacto da programação genética com Pareto em relação à forma clássica no tratamento do efeito *bloat*, diversos testes comparativos foram realizadas para problemas de regressão simbólica. Todos os testes foram executados sob o mesmo conjunto de parâmetros:

- ❑ **População:** 500 indivíduos;
- ❑ **Taxa de cruzamento:** 85%;
- ❑ **Reprodução:** 10%;
- ❑ **Taxa de mutação:** 5%;

- ❑ **Casos de teste:** 100 casos aleatoriamente escolhidos dentro do domínio do problema;
- ❑ **Conjunto de funções:** $+$, $-$, $*$, $/$, $\sin$, $\cos$;
- ❑ **Conjunto de variáveis:** x, y , de acordo com a quantidade de entradas das amostras.
- ❑ **Conjunto de constantes:** criadas ao longo da evolução.

Os problemas tratados foram as regressões simbólicas das funções descritas nas Equações (12), (13) e (14). Elas foram submetidas às duas versões de programação genética – clássica e a com Pareto. Para cada geração, as informações relativas ao tamanho médio das árvores que compõem a população e o erro (aptidão) do melhor indivíduo foram utilizados para a criação de gráficos comparativos, apresentados nas Figuras 22, 23 e 24, respectivamente.

$$2x^3 - 5x + 8, \text{ com } x \in [-1, 1] \quad (12)$$

$$6 \sin x + \cos y, \text{ com } x, y \in [0, 6] \quad (13)$$

$$x^4 + x^3 + x^2 + x, \text{ com } x \in [-2, 2] \quad (14)$$

Os gráficos refletem o impacto causado pela abordagem multiobjetivo, que consegue manter o tamanho médio da população em níveis quase constantes, sem sofrer o crescimento exponencial observado na abordagem clássica. Deve-se levar em consideração que o controle no tamanho médio da população, da forma realizada nestes experimentos, não interferiu negativamente na evolução qualitativa da população. Isso pode ser constatado nas Figuras 25, 26 e 27, que mostram o erro do melhor indivíduo de cada geração para as duas versões da programação genética.

A consideração dos dois objetivos conduziu a direção das buscas para regiões do espaço onde se encontravam os resultados mais relevantes, fazendo com que o tamanho médio dos indivíduos da população se mantivesse quase constante ao longo das gerações. Populações com árvores menores implicam em menor demanda por esforço computacional no momento da avaliação da aptidão e menor quantidade de memória física para acomodá-la. A combinação de todos esses fatores levam a um sistema de programação genética mais eficiente.

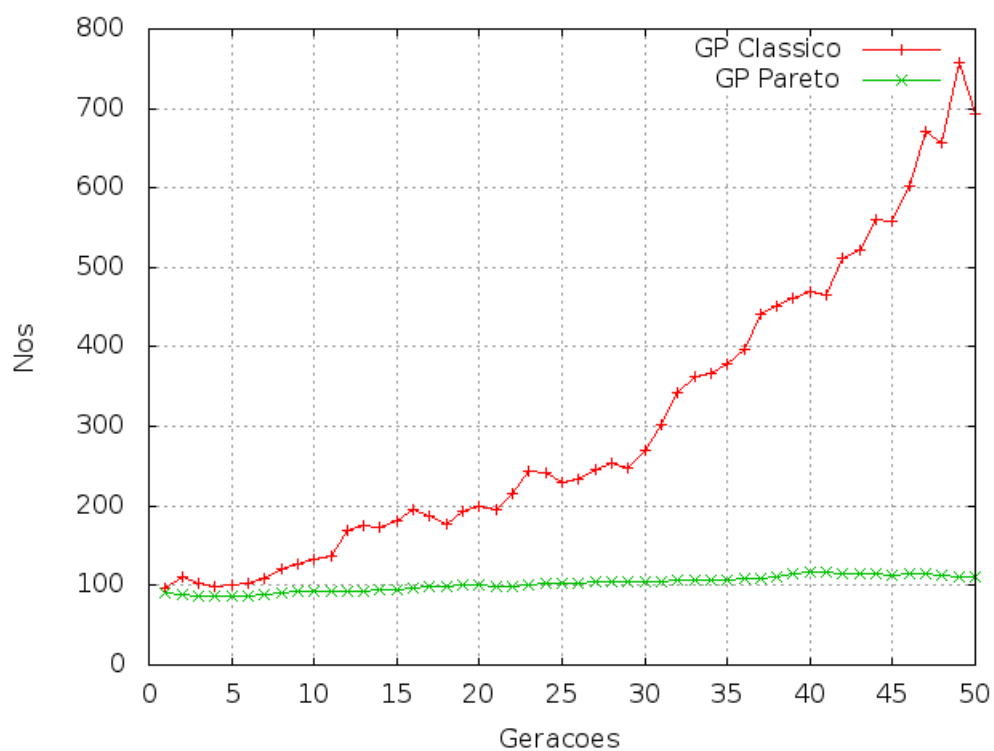


Figura 22 – Crescimento do tamanho médio da população para a função $2x^3 - 5x + 8$.

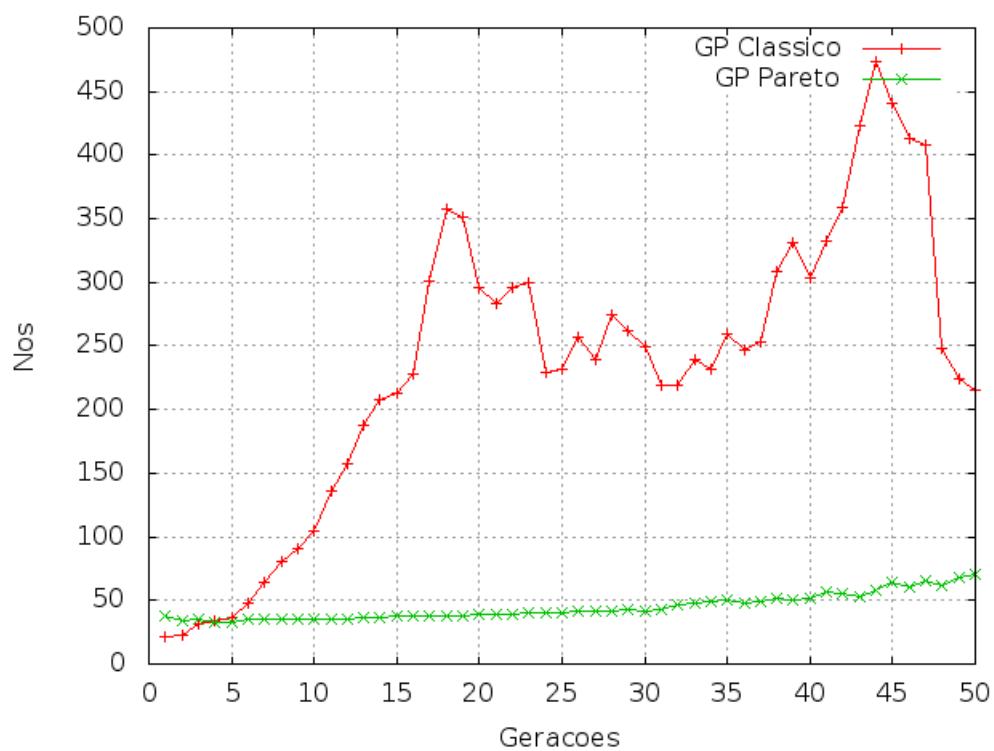


Figura 23 – Crescimento do tamanho médio da população para a função $6 \sin x + \cos y$.

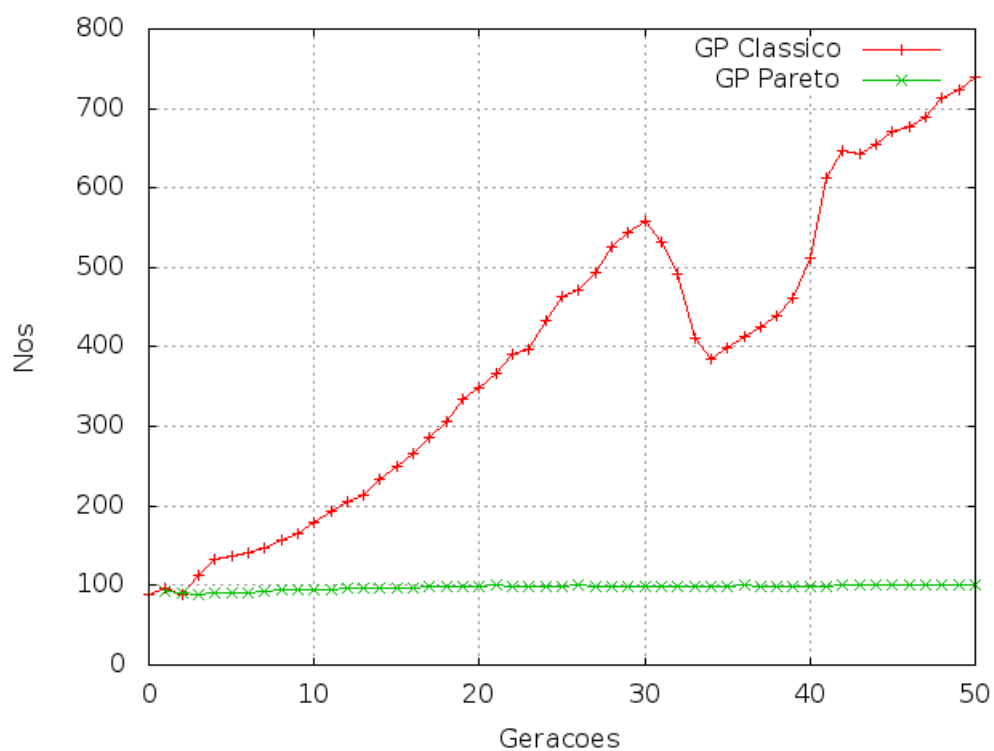


Figura 24 – Crescimento do tamanho médio da população para a função $x^4 + x^3 + x^2 + x$.

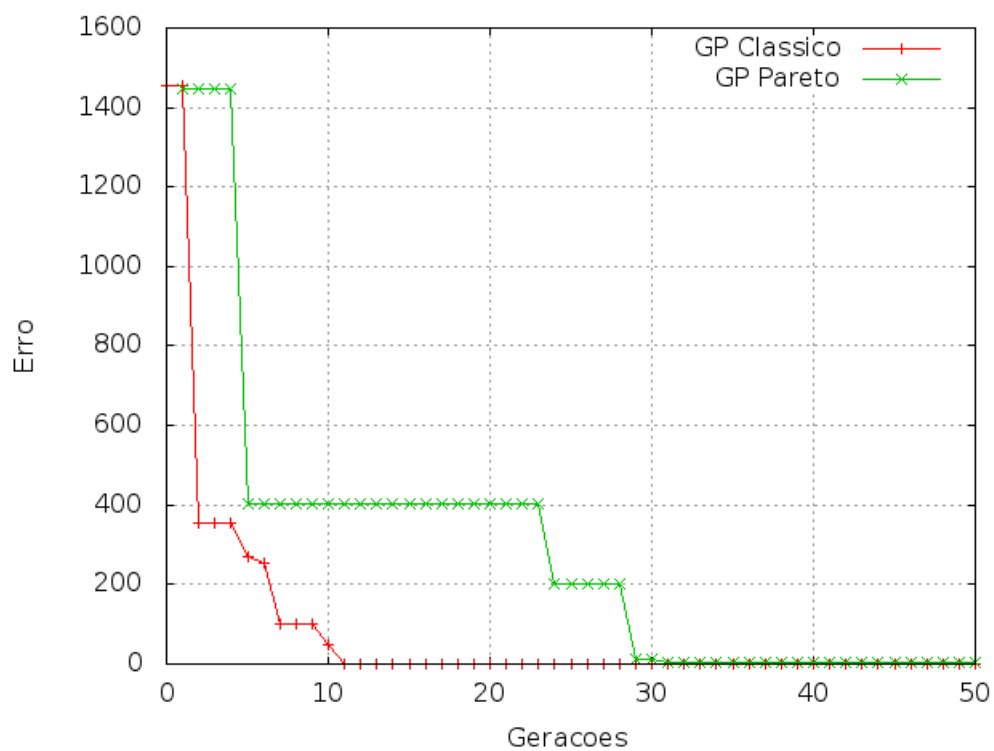


Figura 25 – Valor do erro do melhor indivíduo de cada geração para a função $2x^3 - 5x + 8$.

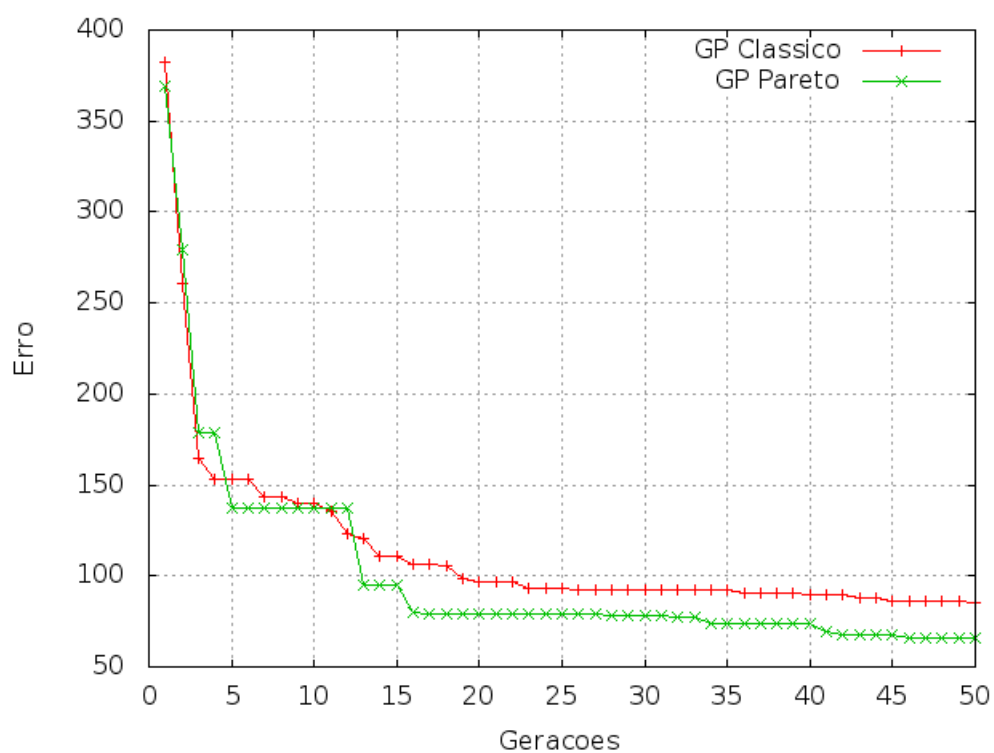


Figura 26 – Valor do erro do melhor indivíduo de cada geração para a função $6 \sin x + \cos y$.

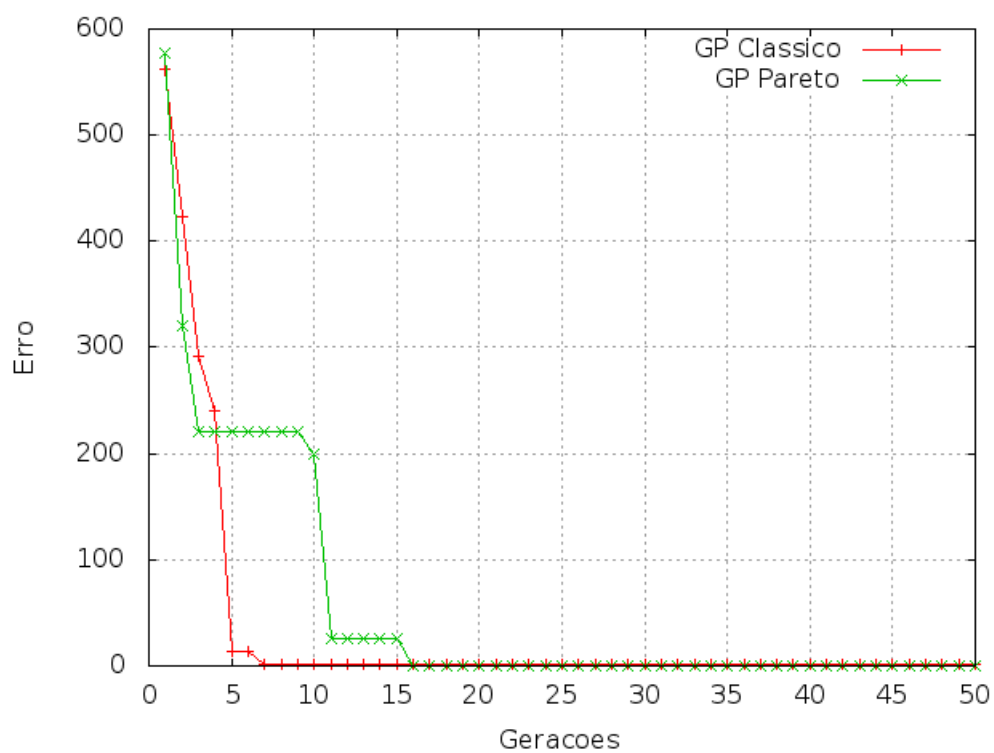


Figura 27 – Valor do erro do melhor indivíduo de cada geração para a função $x^4 + x^3 + x^2 + x$.

Programação genética paralela em processadores *multicore*

A programação genética é uma poderosa técnica evolutiva comumente utilizada para resolver problemas de modelagem e regressão. As dificuldades inerentes à implementação de GP, que requer a manipulação de complexas estruturas de dados e a alta demanda de processamento, podem tornar seu uso problemático e, por vezes, inviável para a maioria dos profissionais sem uma sólida formação em programação. Várias ferramentas, como os *framework* *JGAP* (<http://jgap.sourceforge.net/>) e *EpochX* (<http://www.epochx.org>), foram propostas para minimizar as dificuldades de implementação, mas, pouco foi feito no quesito eficiência.

Por muito tempo, as implementações paralelas de algoritmos evolutivos estiveram restritas aos caros sistemas de arquitetura paralela, com seus múltiplos processadores espalhados em máquinas físicas distintas ou compartilhando a mesma memória física. Recentemente, porém, a popularização de processadores *multicore* a preços acessíveis tem possibilitado uma melhor experiência em termos de poder de processamento.

Este capítulo apresenta a implementação de uma ferramenta multiplataforma que utiliza o modelo de ilhas em programação genética otimizado para processadores *multicore*. Tal ferramenta é aplicada a um problema de regressão simbólica, a fim de medir o tempo de execução das versões paralela e sequencial.

4.1 A alta demanda por poder de processamento da GP

A aplicação dos operadores genéticos em uma população (como *crossover*, mutação, reprodução, etc.), demanda tempo constante e, quase sempre, desprezível. Logo, tais operadores pouco influenciam no tempo total gasto na evolução via programação genética. Entretanto, o mesmo não pode ser dito da avaliação dos indivíduos para mensurar suas

aptidões.

A avaliação das aptidões de todos os indivíduos de uma população é um processo que pode incluir várias etapas: o caminhar em árvore, a execução do programa por um compilador externo, a adequação do indivíduo para que este possa ser submetido a um simulador, etc. Tudo isso é agravado pela quantidade de casos de teste que, para problemas reais, tende a ser alta.

Populações grandes podem também comprometer o uso de memória do sistema. Para os algoritmos genéticos, que utilizam estruturas de tamanho fixo, como vetores de inteiros, o armazenamento pode não ser um problema. O mesmo, no entanto, não pode ser dito da programação genética, que trabalha com estruturas de árvores que podem assumir diferentes formas e tamanhos e requerer grandes áreas de memória. Por essa razão, o controle de *bloat* (conforme visto no Capítulo 3) é necessário para tornar a execução um processo viável.

4.2 Abordagem paralela

Conforme discutido na Seção 2.8, o tamanho da população está entre os parâmetros mais importantes no que se refere à programação genética. Quanto mais complexo for o problema, maior será a quantidade de indivíduos que devem evoluir. Nesta categoria incluem-se a maior parte dos problemas de cunho prático.

O crescimento da população é acompanhado pelo aumento do poder de processamento demandado. A estratégia mais comumente utilizada é a paralelização, ou seja, a quebra do problema em vários subproblemas que possam ser simultaneamente executados em unidades de processamento distinto, que podem ser:

- ❑ processadores localizados em máquinas diferentes interligadas via rede;
- ❑ um ou mais processadores localizados em uma mesma máquina, cada um com um ou mais *cores*;
- ❑ alguma combinação das opções anteriores.

Das opções listadas, a que tem ganhando maior destaque é a utilização de processadores com múltiplos núcleos – *multicore*. Esses dispositivos são atualmente encontrados a preços acessíveis, mas seus benefícios só podem ser efetivamente alcançados através de metodologias específicas de programação, que muitas vezes incluem análises de algoritmos complexos, a fim de identificar os segmentos de código que podem ser efetiva e eficientemente executados de forma concorrente.

4.3 O modelo de ilhas

O campo de paralelização em algoritmos evolutivos tem recebido diferentes enfoques nas últimas décadas. Em Koza (1992a) e Goldberg (1989), por exemplo, são sugeridos modelos que paralelizam a base de indivíduos da população, os casos de teste ou ainda modelos de execuções independentes.

Dentre os vários modelos possíveis de paralelização, destacam-se:

- ❑ Modelo mestre-escravo, em que um computador central faz o gerenciamento de um conjunto de outros computadores (OUSSAIDÈNE et al., 1996);
- ❑ Modelo de difusão, em que a população é distribuída por unidades de processamento cuja população evolui separadamente e seus indivíduos trocam material genético apenas com os seus vizinhos (numa topologia estabelecida a priori), o que gera, ao longo das gerações, regiões de aptidões semelhantes (DUMITRESCU et al., 2000); e
- ❑ Modelo de ilhas, visto mais aprofundadamente nesta seção.

No modelo de ilhas, há a alocação de grandes subpopulações (chamadas também de *demes*) em nós de processamento separados. Quando a execução começa, subpopulações locais são randomicamente criadas em cada nó e as aptidões dos indivíduos são localmente calculadas, dando início ao processo de evolução. Como os cálculos das aptidões são realizados de forma independente em cada subpopulação, essa abordagem de paralelização tende a aumentar o desempenho em razão aproximadamente linear no número de nós de processamento (ANDRE; KOZA, 1996).

A evolução isolada das subpopulações em cada ilha pode levar à convergência prematura e à restrição do espaço de busca em ótimos-locais. Para evitar que isso ocorra, o modelo de ilhas prevê também o esquema de migração, que consiste no envio (cópia) dos n melhores indivíduos da população de uma ilha para substituir os n piores indivíduos da população de outra ilha.

A frequência com que a migração ocorre é, geralmente, definida por um intervalo entre gerações – 10 gerações é o mais comum na literatura – e várias são as topologias de vizinhança possíveis. A migração é de fundamental importância neste modelo, por proporcionar saltos evolucionários e preservar a diversidade populacional de todo o sistema (TOMASSINI, 2005). A migração requer também uma topologia organizada em vizinhança. A forma mais usual, também utilizada neste trabalho, é a topologia em anel, vista na Figura 28.

Por fim, se em qualquer momento da evolução um indivíduo que atenda completamente aos requisitos estabelecidos pelo problema for encontrado em qualquer subpopulação, todos os demais nós de processamento são notificados e a execução do programa encerrada. Caso esse indivíduo não seja encontrado, cada ilha continuará sua evolução

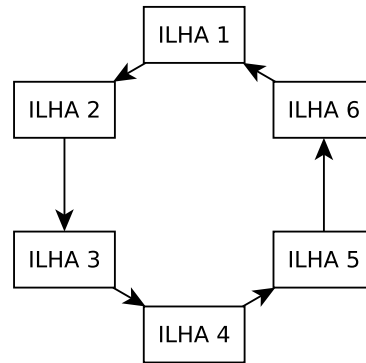


Figura 28 – Topologia de migração em anel. Cada ilha envia e recebe indivíduos. Neste esquema, isso se dá obedecendo o sentido anti-horário.

até a quantidade máxima de gerações ser alcançada. Os melhores indivíduos de cada ilha são comparados para se obter a resposta do sistema (ANDRE; KOZA, 1996).

4.4 Modelo de ilhas em programação genética com Pareto

No Capítulo 3, foi descrita a abordagem de programação genética multiobjetivo (programação genética com Pareto) para controle de *bloat*. As pesquisas realizadas na etapa de levantamento bibliográfico não apontaram qualquer método capaz de unir os conceitos de dominância com o paralelismo em ilhas. Por essa razão, criou-se neste trabalho uma metodologia para a escolha dos indivíduos da ilha que serão enviados para a ilha vizinha e também para aqueles que serão substituídos.

Conforme foi visto, há sempre duas populações: P_t , que preserva os melhores indivíduos encontrados até a geração t , e Q_t , que abriga os indivíduos gerados pela aplicação dos operadores genéticos aos indivíduos de P_t . Para selecionar os n indivíduos que serão enviados para outras ilhas, P_t e Q_t são unidas no mesmo conjunto R_t , cujos elementos são, em seguida, separados em suas fronteiras de dominância $\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \dots)$. Os indivíduos são selecionados dentro das fronteiras de dominância, iniciando-se da primeira, até que a quantidade n seja atingida. Processo semelhante ocorre na ilha que receberá os emigrantes: a população desta ilha é também separada em suas fronteiras de dominância e os elementos que compõem as últimas fronteiras são selecionados para serem substituídos.

Embora não haja garantia de que as soluções não dominadas de uma ilha pertençam à fronteira de ótima de Pareto em outra, a introdução de novos indivíduos pode auxiliar na exploração de novos espaços de busca até então desconhecidos.

4.5 Processadores *multicore* e programação concorrente

4.5.1 Arquitetura básica de computadores

Ao conjunto de unidades funcionais que compõem a arquitetura básica de um computador, dá-se o nome de unidade básica de processamento, ou *Central Processing Unit* (CPU). De acordo com Stallings (2002), cinco são as ações básicas atribuídas à CPU:

- ❑ **Buscar instruções** da memória;
- ❑ **Interpretar** as instruções;
- ❑ **Buscar dados** da memória ou de dispositivos de entrada/saída;
- ❑ **Processar dados** por meio da execução de operações lógicas e aritméticas sobre eles; e
- ❑ **Escrever dados**, pois o resultado da execução pode requerer escrita na memória ou em dispositivos de entrada/saída.

A Figura 29 apresenta o esquema básico de uma CPU. A unidade de controle determina a sequência em que as instruções devem ser executadas. A unidade lógica e aritmética executa as instruções do programa. Tais instruções são trazidas dos registradores (memórias pequenas e velozes) por meio do barramento interno.

4.5.2 *Threads* e *Hyper-Threading*

Threads são linhas de execução separadas, cada uma contendo sua própria pilha de chamadas de método e seu próprio contador de programa. Várias *threads* podem ser executadas simultaneamente, com todas compartilhando recursos no nível de aplicativo – como memória (DEITEL; DEITEL, 2009).

Se um computador é dotado de um processador com um único núcleo e nele são executadas múltiplas *threads*, então, por meio da troca rápida de programas em execução (multitarefa) há a impressão de que várias linhas de execução ocorrem simultaneamente. A empresa *Intel* criou a tecnologia *Hyper-Threading*, que permite, em processadores com um único núcleo, a execução simultânea de dois programas sem realizar a troca de contexto entre tarefas. Nessa situação, o sistema operacional se comporta como se possuísse dois processadores disponíveis mas, por limitações principalmente relacionadas à disponibilidade de recursos compartilhados, um processador com *Hyper-Threading* sempre tem desempenho menor do que os processadores com dois núcleos reais, que permitem o verdadeiro paralelismo (AKHTER; ROBERTS, 2006).

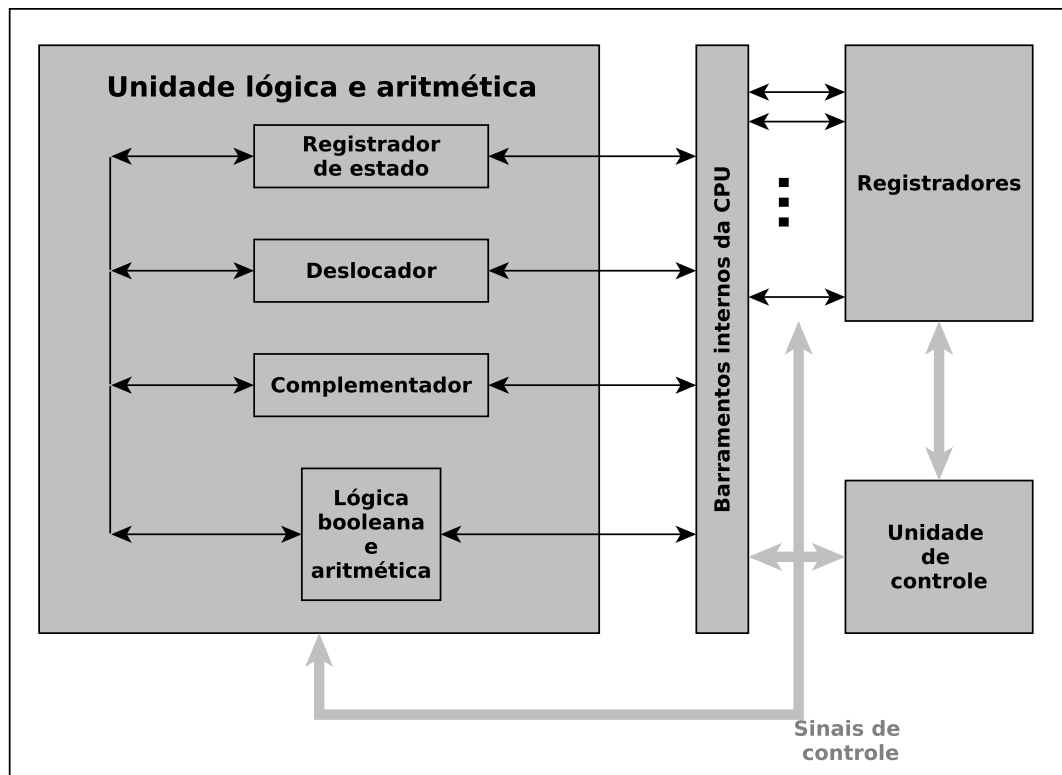


Figura 29 – Unidade central de processamento, com suas componentes básicas (unidade lógica e aritmética, unidade de controle e registradores) e suas interconexões. Adaptado de Stallings (2002).

4.5.3 Tecnologia *multicore*

Desde a década de 1960, quando os primeiros computadores de propósito geral foram fabricados, o desempenho dos processadores aumentou substancialmente. Entretanto, este aumento tem encontrado, desde 2002, barreiras difíceis (ou mesmo impossíveis) de superar. Processadores mais potentes consomem mais energia e este consumo aumenta a quantidade de calor dissipado e complica a qualidade do projeto. No que se refere à memória, mesmo que o processador consiga ter acesso a maior quantidade de memória física, o acesso a ela chega a ser 50 vezes mais lento do que uma operação aritmética de multiplicação. Por fim, a otimização realizada internamente nos processadores para permitir a execução de instruções simultâneas parece ter chegado ao limite.

Devido a essas barreiras, em 2004 a empresa *Intel*, em parceria com a *IBM* e à *Sun*, começou a investir na fabricação em massa de processadores com múltiplos núcleos de processamento (do inglês *core*), tecnologia que ficou conhecida por *multicore*. Cada núcleo executa com frequência de *clock* mais baixa, dissipando menos calor do que processadores de núcleo único com velocidade equivalente.

O suporte ao paralelismo real faz com que os processadores *multicore* tragam grandes

benefícios a programas com múltiplas linhas de execução. Porém, nem todas as linguagem mais populares (como C e C++) trazem recursos nativos para criar aplicações paralelas (DEITEL; DEITEL, 2009). A linguagem Java implementa nativamente o conceito de programação *multithreading* desde as suas primeiras versões. Em sua última edição, um novo *framework* foi oferecido para dar suporte a aplicações paralelas de forma relativamente simples, conforme apresentado a seguir.

4.5.4 O *framework fork/join*

Embora os processadores *multicore* apresentem grande poder de processamento, uma aplicação só aproveitará essa vantagem se for adequadamente programada. A linguagem Java, em sua sétima edição, traz um recurso para este fim: o *framework fork/join*, uma implementação de `ExecutorService` que, como todas as demais implementações desta interface, trabalha sob um *pool* de *threads*, mas que se diferencia por estar baseada no algoritmo *work-stealing*, que melhor gerencia a distribuição das tarefas e consegue obter um maior aproveitamento dos núcleos de processamento disponíveis.

O *framework fork/join* trabalha tratando as tarefas de maneira recursiva, conforme apresentado no Algoritmo 8. Estando o problema escrito na forma do Algoritmo 8, este deve ser colocado em uma classe que implemente uma das interfaces de `ForkJoinTask`. Duas são as opções: a `RecursiveTask`, que pode retornar valores, e a `RecursiveAction`. A instância de `RecursiveTask` ou `RecursiveAction` que representa o problema como um todo deve ser passada para o método `invoke()` de `ForkJoinPool`. No momento da instanciación do objeto `ForkJoinPool`, o número de *cores* disponíveis pode ser informado.

Algoritmo 8 Lógica de funcionamento do *framework fork/join*

- 1: **se** a tarefa é suficientemente pequena **então**
 - 2: resolva a tarefa
 - 3: **senão**
 - 4: quebre a tarefa em duas partes
 - 5: aplique este algoritmo às duas partes e aguarde o resultado
 - 6: **fim se**
-

Algumas particularidades do uso desse *framework* são reunidas no Apêndice A, onde é dado o exemplo de uma aplicação. Uma referência completa pode ser obtida no endereço <http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>, que contém a documentação oficial da linguagem.

4.6 Implementação

A implementação foi desenvolvida em linguagem Java e utilizou o *framework fork/join*, visto na Seção 4.5.4, e adotou o modelo de paralelismo em ilhas, visto na Seção 4.3, com

a topologia de migração em anel (Figura 28). A implementação da GP é baseada em árvore, com a abordagem multiobjetivo tratada no Capítulo 3.

A aplicação foi implementada de maneira adaptativa: ela obtém a informação referente à quantidade n de núcleos de processamento existentes no computador em uso e cria n ilhas. A população é uniformemente distribuída pelas ilhas e a migração ocorre a cada dez gerações. A quantidade de indivíduos escolhidos para migrar são obtidos nas primeiras fronteiras de dominância, conforme explicado na Seção 4.4, não devendo ultrapassar 10% da subpopulação.

Deve-se observar que, dependendo da abordagem utilizada, a comunicação entre os processos paralelos (ilhas) pode comprometer o desempenho geral do sistema. Na implementação aqui apresentada, a comunicação é feita sem que haja o bloqueio (e conseqüente atraso) na execução dos processos. A cada 10 gerações, a ilha i disponibiliza indivíduos para a ilha $i + 1$ e estes são armazenados em um *buffer* para aguardarem o momento em que a ilha $i + 1$ solicitará o ingresso de novos indivíduos. Após a disponibilização, a ilha i continua seu processamento normalmente. Também a cada 10 gerações, cada ilha verifica se há indivíduo em seu *buffer*. Caso haja, eles são inseridos, substituindo os n piores indivíduos atuais; caso contrário, o processamento segue sem interferências.

O trecho de código a seguir ilustra como foi realizado o paralelismo via *framework fork/join*.

```
private class ForkJoinPG extends RecursiveAction {
    private int inicio, fim;
    private int tamSubPop;

    public ForkJoinPG(int inicio, int fim) {
        this.inicio = inicio;
        this.fim = fim;
    }
    @Override
    protected void compute() {
        tamSubPop = fim - inicio + 1;
        if (tamSubPop <= subPop) {
            int ilha = (int) ((inicio + fim) / 2) / subPop;
            executaPG(ilha);
            return;
        }
        int meio = (fim + inicio) / 2;
        ForkJoinTask t1 = new ForkJoinPG(inicio, meio);
        ForkJoinTask t2 = new ForkJoinPG(meio + 1, fim);
        invokeAll(t1, t2);
    }
}
```

A classe interna `ForkJoinPG` implementa a interface `RecursiveAction` e recebe como argumento os valores `inicio` e `fim`. Em sua primeira execução, `inicio` recebe o valor zero e `fim`, o tamanho da população menos 1. O valor obtido pelo cálculo de `fim - inicio +`

1 corresponde ao tamanho da subpopulação representada neste intervalo. Caso este valor seja menor ou igual ao tamanho de subpopulação ideal (obtido pela divisão do tamanho da população pelo número de processadores e representado pela variável `subPop`), o método de programação genética é chamado para a ilha correspondente. Caso contrário, a tarefa é quebrada em subpopulações menores e novas instâncias de `ForkJoinPG` são criadas recursivamente.

Ao término do processamento de todas as ilhas, todas as subpopulações são unidas em uma única população e esta é submetida ao *Fast non-dominated sort* (Algoritmo 5). Assim, os melhores indivíduos de todas as populações encontrar-se-ão na fronteira ótima de Pareto $\mathcal{F}_1$, de onde é retirada a saída do programa.

4.7 Resultados e discussões

Para medir a diferença de tempo de processamento entre as versões paralela e não paralela, ambas foram executadas, sob os mesmos parâmetros, para resolver o mesmo problema de regressão simbólica. O resultado é dado no gráfico da Figura 30.

Nesse gráfico são mostradas as médias de tempo obtidas na execução do programa em um computador equipado com um processador Intel Core i5-2500, 3.30GHz, que possui quatro núcleos físicos, instalado em um ambiente com 3,8 GB de memória física, e sistema operacional Ubuntu 13.04 64 bits. Os dados foram obtidos pela média das tomadas de tempo (em nanossegundos) de 100 execuções da versão sequencial e 100 execuções da versão paralela. Os testes foram realizados utilizando 50 pares (x, y) da equação $y = x^4 + x^3 + x^2 + x$ aleatoriamente escolhidos.

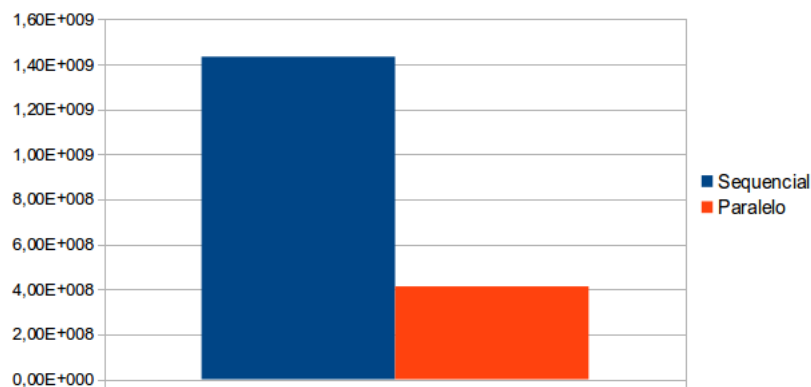


Figura 30 – O gráfico mostra a eficiência da versão paralela de programação genética quando comparada à versão sequencial tradicionalmente utilizada.

As tomadas de tempo mostraram que a execução da versão paralela resultou em um tempo 3,5 vezes menor do que na versão sequencial, tradicionalmente utilizada. Esta diferença é o resultado da distribuição da população em quatro subpopulações, cada uma

associada a um único núcleo do processador e evoluindo de maneira paralela e independente, estando de acordo com o comportamento descrito por Andre e Koza (1996).

A versão sequencial pouco difere das implementações já conhecidas, como o *JGAP*, o *EpochX*, dentre outros *framework* livremente distribuídos, já que todos implementam os algoritmos clássicos de programação genética em árvore. O acréscimo na eficiência, na versão aqui apresentada, se deve à adoção do modelo de ilhas adaptado ao ambiente provido de processadores com múltiplos núcleos.

Por ocultar os detalhes de implementação do paralelismo, a ferramenta desenvolvida é de fácil manipulação: todo o processamento e manutenção da execução concorrente nos núcleos disponíveis é transparente ao usuário. Embora o *JGAP* também implemente o modelo de ilhas, isto é feito através de conexões de rede, espalhando as ilhas por máquinas unidas pelo protocolo TCP/IP, o que requer um ambiente próprio composto por máquinas apropriadamente configuradas, nem sempre uma tarefa simples de se realizar.

Além de conseguir bons resultados na obtenção das equações, a implementação aqui apresentada mostrou-se superior à versão tradicional de GP, que é sequencial, por melhor aproveitar as características dos processadores compostos por múltiplos núcleos. Com a popularização de equipamentos deste tipo, a programação genética paralela não mais ficará restrita aos caros ambientes de computação distribuída de alto desempenho.

Uma ferramenta para programação genética paralela com Pareto

A programação genética, conforme discutido no Capítulo 2, apresenta grande potencial de aplicação nas mais diferentes áreas. Porém sua implementação demanda conceitos avançados de programação, o que restringe bastante sua utilização.

Uma ferramenta que implemente os principais aspectos da programação genética e que acrescente os requisitos de qualidade dos resultados (Capítulo 3) e eficiência (Capítulo 4) pode ser de grande utilidade profissionais de diferentes linhas. Assim, este capítulo apresenta a *Parallel Pareto Genetic Programming* (PPGP), uma ferramenta capaz de atender a estas demandas e desenvolvida para atender a dois públicos: programadores experientes com ou sem familiaridade com computação evolutiva, que podem utilizá-la como uma *Application Programming Interface* (API), e profissionais sem conhecimento em programação, mas que querem realizar modelagem de dados via regressão simbólica utilizando a ferramenta como um aplicativo.

5.1 A ferramenta para desenvolvedores

A ferramenta desenvolvida recebeu o nome de PPGP e traz a implementação dos principais conceitos de programação genética expandidos aos temas de dominância de Pareto, para garantir a qualidade das respostas geradas, e de paralelização em processadores *multicore*, para trazer ganhos no tempo de execução.

A PPGP, conforme visto nesta seção, é de fácil utilização e não requer conhecimentos avançados de programação ou de computação evolutiva. Espera-se apenas que usuário conheça os fundamentos da linguagem Java. Todo o fluxo de utilização pode ser resumido nos seguintes passos:

1. Definir o conjunto de treinamento;
2. Criar o objeto PG com os dados de conjunto de treinamento;

3. Definir em PG os conjuntos de funções e constantes;
4. Executar o programa;
5. Recolher, de PG, os resultados da execução (estrutura gerada, gráficos, tomadas de tempo, etc.).

A usuários com maiores conhecimentos em programação genética é dada a possibilidade de modificar os parâmetros *default*, como tamanho da população, quantidade de gerações, probabilidades e tipos de cruzamento e mutação, probabilidade de reprodução, etc. Já aqueles que possuem mais familiaridade com programação estão livres para criar novas funções que sejam mais adequadas ao domínio do problema em análise.

5.1.1 Conjunto de amostras

A ferramenta é capaz de mapear vetores de $\mathbb{R}^n \rightarrow \mathbb{R}^m$, podendo, assim, tratar problemas com:

- ❑ uma entrada e uma saída;
- ❑ uma entrada e várias saídas;
- ❑ várias entradas e uma saída;
- ❑ várias entradas e várias saídas.

O conjunto de treinamento (casos de *fitness*), que apresenta o comportamento que se deseja reproduzir, é passado à ferramenta por meio de matrizes de entrada/saída ou por arquivos-texto, com valores numéricos inteiros ou reais. Em ambos os casos, os conjuntos devem ser passados como argumentos do construtor da classe PG.

A primeira forma é simples, porém pouco eficaz em situações em que o conjunto de treinamento é muito grande. As matrizes devem ser do tipo *double* e são passadas ao objeto da classe PG da seguinte forma:

```
double [][] entradas = new double [][] {{1, 0, 0},
                                           {1, 0, 1},
                                           {1, 1, 0},
                                           {1, 1, 1}};

double [][] saidas = new double [][] {{1, 0, 0, 0},
                                       {0, 1, 0, 0},
                                       {0, 0, 1, 0},
                                       {0, 0, 0, 1}};

PG pg = new PG(entradas, saidas);
```

Caso se opte pelo uso de arquivos, estes devem ser construídos seguindo certos padrões. Duas são as possibilidades: dois arquivos, um com as entradas e outro com as saídas, ou um único arquivo, com ambas as informações.

```
PG pg = new PG("/home/leonardo/arquivos/entradas.txt", "/home/leonardo/arquivos/saídas.txt");
```

```
PG pg = new PG("/home/leonardo/arquivos/multiplex1.txt");
```

O uso de um único arquivo apresenta algumas particularidades. Quando nenhuma informação adicional está presente, considera-se, para cada caso de teste (cada linha do arquivo), que os $n - 1$ primeiros elementos são as entradas e o n -ésimo a saída. Se esta não for a configuração desejada, a primeira linha do arquivo deve conter um inteiro k com o número de entradas. Assim, os k primeiros elementos de cada linha serão as entradas e os $n - k$ restantes, as saídas. Um arquivo formatado desse modo é mostrado na Tabela 3.

Tabela 3 – Exemplo de arquivo com padrões de treinamento para um multiplexador com três entradas e quatro saídas.

3						
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

5.1.2 Primitivas

Após a criação do objeto do tipo **PG**, deve-se informar quais primitivas serão utilizadas como genes dos indivíduos cuja população será evoluída. Conforme foi discutido na Seção 2.3, tais primitivas são funções, constantes e variáveis. Estas são tratadas pela ferramenta da seguinte maneira:

- ❑ As **funções** são o único tipo de primitiva que deve ser, obrigatória e explicitamente, informado pelo usuário. Há por padrão um conjunto de funções pré-definidas disponíveis para o uso, mas novas definições que melhor se adequem ao problema em análise podem ser facilmente implementadas.
- ❑ As **constantes** são valores numéricos reais. Embora seja útil em certos tipos de problemas (como o valor da aceleração da gravidade em problemas relacionados a movimento em sistemas físicos, por exemplo), nem sempre há um consenso com relação a quais valores são os ideais. Por essa razão, seu fornecimento é opcional.
- ❑ As **variáveis** são implicitamente informadas, uma vez que a simples análise do conjunto de treinamento por parte da ferramenta já é capaz de inferir a quantidade de variáveis que constará no problema. Simbolicamente, as variáveis são representadas por $x, y, z, u, v, w, r, s, t$.

Tabela 4 – Primitivas disponíveis.

Função	Classe (.java)	Label
Adição	Soma	+
Subtração	Subtracao	−
Multiplicação	Multiplicacao	*
Divisão protegida	Divisao	/
Seno	Seno	sin
Cosseno	Cosseno	cos
Tangente	Tangente	tan
Exponencial (e^x)	Exp	exp
Logaritmo natural	LogaritmoN	$\ln$
Logaritmo comum	Logaritmo10	log
Raiz Quadrada	RaizQuadrada	$\sqrt{}$
Mínimo	Min	$\min$
Máximo	Max	$\max$
Conjunção	AND	<i>and</i>
Disjunção	OR	<i>or</i>
Negação	NOT	<i>not</i>
Ou exclusivo	XOR	<i>xor</i>
Não E	NAND	<i>nand</i>
Não OU	NOR	<i>nor</i>

5.1.2.1 Adicionando funções pré-definidas

A PPGP traz um conjunto de funções pré-definidas, mostradas na Tabela 4. Este conjunto cobre as funções aritméticas e trigonométricas básicas, funções exponencial, logarítmicas, lógicas, dentre outras.

Para informar ao objeto PG quais funções serão utilizadas, uma das abordagens abaixo deve ser utilizada:

- ❑ Criando e adicionando uma lista de funções:

```
...
ArrayList<IFuncao> portas = new ArrayList<>();
portas.add(new AND());
portas.add(new NAND());
portas.add(new NOR());
portas.add(new NOT());
portas.add(new OR());
portas.add(new XOR());

pg.addFuncao(portas);
...
```

- ❑ Criando e adicionando um vetor de funções:

```
...
IFuncao[] portas = new IFuncao[6];
```

```
portas[0] = new AND();
portas[1] = new NAND();
portas[2] = new NOR();
portas[3] = new NOT();
portas[4] = new OR();
portas[5] = new XOR();

pg.addFuncao(portas);
...
```

❑ Adicionar as funções diretamente ao objeto PG:

```
...
pg.addFuncao(new AND());
pg.addFuncao(new NAND());
pg.addFuncao(new NOR());
pg.addFuncao(new NOT());
pg.addFuncao(new OR());
pg.addFuncao(new XOR());
...
```

A escolha das funções é realizada com base no tipo de problema que se deseja trabalhar. Nos exemplos dados nas linhas de código acima, são manipuladas funções booleanas, úteis na criação de circuitos digitais combinacionais.

5.1.2.2 Definição de novas funções

É natural que o conjunto de funções *default* não seja suficiente para cobrir todos os problemas que podem ser tratados com programação genética, podendo, assim, haver a necessidade de implementar novas funções.

A ferramenta permite a criação de qualquer tipo de função que receba uma quantidade arbitrária de argumentos e que retorne um valor real. Para tanto, basta criar uma classe que obedeça à seguinte interface:

```
package pg.arvore.no.interfaces;

public interface IFuncao {
    public double opera(double ... a);
    public int getAridade();
    public String toString();
}
```

O método `opera` é o responsável por receber os argumentos, efetuar os cálculos necessários e retornar o resultado. A notação `(double ... a)` indica que, na implementação do método abstrato, a definição de qualquer quantidade de argumentos é aceita. Esta quantidade deve ser informada pelo método `getAridade`. O *label* da função deve ser informado no método `toString`.

Para exemplificar o que foi dito, temos abaixo a implementação da classe `Multiplicação`, que recebe dois valores reais e retorna seu produto:

```
public class Multiplicacao implements IFuncao{

    @Override
    public double opera(double... a) {
        return a[0]*a[1];
    }

    @Override
    public int getAridade() {
        return 2;
    }

    @Override
    public String toString(){
        return "*";
    }
}
```

Os argumentos do método `opera` encontram-se dentro do vetor `a`. Como é uma função de apenas dois argumentos, o valor inteiro 2 é retornado de `getAridade`. Seu label é `*`.

Esta nova função deve ser colocada no pacote `aplicacoes.rsimbol.src.funcoes`. Uma vez definida, ela pode ser adicionada ao objeto PG da mesma forma que foi mostrado na Seção 5.1.2.1.

5.1.2.3 Adicionando constantes

As constantes, podem ser facilmente adicionadas ao objeto PG por meio do comando `addConstante`:

```
pg.addConstante(10);
```

Pode-se passar ao objeto um único valor real ou um conjunto de valores, organizadas em forma de um vetor unidimensional ou ainda por um lista do tipo `ArrayList<Double>`.

5.1.3 Operadores genéticos disponíveis

Por se tratar de uma ferramenta de propósito geral, há uma ampla quantidade de operadores genéticos disponíveis para o uso, tendo em vista que alguns operadores podem ser mais adequados no tratamento de um tipo de problema do que outro.

5.1.3.1 Operadores de recombinação

A ferramenta traz implementada sete tipos de operadores de recombinação. A determinação do tipo de cruzamento é feita por meio do método `setTipoCrossover`, da classe PG, que recebe como argumento um valor constante da classe `Arvore`:

- ❑ PG.KOZA_CROSSOVER: *Crossover* de sub-árvore;
- ❑ PG.ONE_POINT_CROSSOVER: *Crossover* de um ponto;
- ❑ PG.STRICT_ONE_POINT_CROSSOVER: *Crossover* de um ponto restrito;
- ❑ PG.UNIFORM_CROSSOVER: *Crossover* uniforme;
- ❑ PG.STRICT_UNIFORM_CROSSOVER: *Crossover* uniforme restrito;
- ❑ PG.SIZE_FAIR_CROSSOVER: *Size-fair crossover*;
- ❑ PG.RANDOM_CROSSOVER: executa um dos operadores *crossover* disponíveis, escolhido aleatoriamente.

As particularidades de cada operador foram discutidas na Seção 2.7.2, e cabe ao usuário da ferramenta escolher, com base nas informações apresentadas, a variação mais adequada.

5.1.3.2 Operadores de mutação

Semelhante à abordagem efetuada com o operador de recombinação, algumas variações de mutação foram implementadas: a mutação de um ponto, a mutação de subárvore e o *size-fair subtree mutation* (ver Seção 2.7.3). A determinação do tipo de mutação utilizada pela ferramenta é feita pelo repasse de uma constante estática do objeto da classe **Arvore**:

- ❑ PG.SUBTREE_MUTATION: Mutação de subárvore;
- ❑ PG.SIZE_FAIR_SUBTREE_MUTATION: *size-fair subtree mutation*;
- ❑ PG.POINT_MUTATION: mutação de um ponto;
- ❑ PG.RANDOM_MUTATION: executa um dos operadores de mutação disponíveis, escolhido aleatoriamente.

5.1.4 Tipos de função de aptidão

A medida de aptidão do indivíduo pode ser determinante na qualidade de resposta de um problema específico. Por essa razão, a ferramenta traz quatro tipos de função de aptidão, definidas por meio de constantes:

- ❑ Aptidao.MINIMIZA_ERRO
- ❑ Aptidao.MINIMIZA_TERMINAIS
- ❑ Aptidao.MINIMIZA_NAO_TERMINAIS
- ❑ Aptidao.MINIMIZA_ELEMENTOS

A classe PG repassa as matrizes de treinamento para a classe `Aptidao`, assim como o método de avaliação escolhido. Sempre que se desejar saber a aptidão de um indivíduo, basta chamar o método `avalia` para obter um valor numérico real de acordo com a metodologia de avaliação utilizada.

É válido lembrar que esta aptidão é utilizada apenas no final do processo de evolução, para escolher o melhor indivíduo dentro da fronteira ótima de Pareto.

5.1.5 Gráficos disponíveis

Os gráficos são de grande utilidade, pois permitem que o pesquisador analise o comportamento do problema tratado por programação genética e direcione suas ações. Os gráficos presentes na ferramenta são:

- ❑ `salvaGraficoComparacao`: mostra duas curvas: uma com a plotagem dos pontos de treinamento e outra com o gráfico encontrado pela ferramenta – Figura 31;



Figura 31 – Gráfico de comparação.

- ❑ `salvaGraficoTamanhoIlhas`: mostra o tamanho médio das árvores que compõem a população em cada geração. Este é um método sobrecarregado que pode ser usado com argumento (um inteiro que especifica a numeração da ilha) ou sem argumento (apresentando a informação global da população de todas as ilhas) – Figura 32;
- ❑ `salvaGraficoDesempenhoIlhas`: mostra duas curvas: uma com o valor de aptidão do melhor indivíduo em cada geração e outra com a médias das aptidões dos indivíduos em cada geração. Este é um método sobrecarregado que pode ser usado

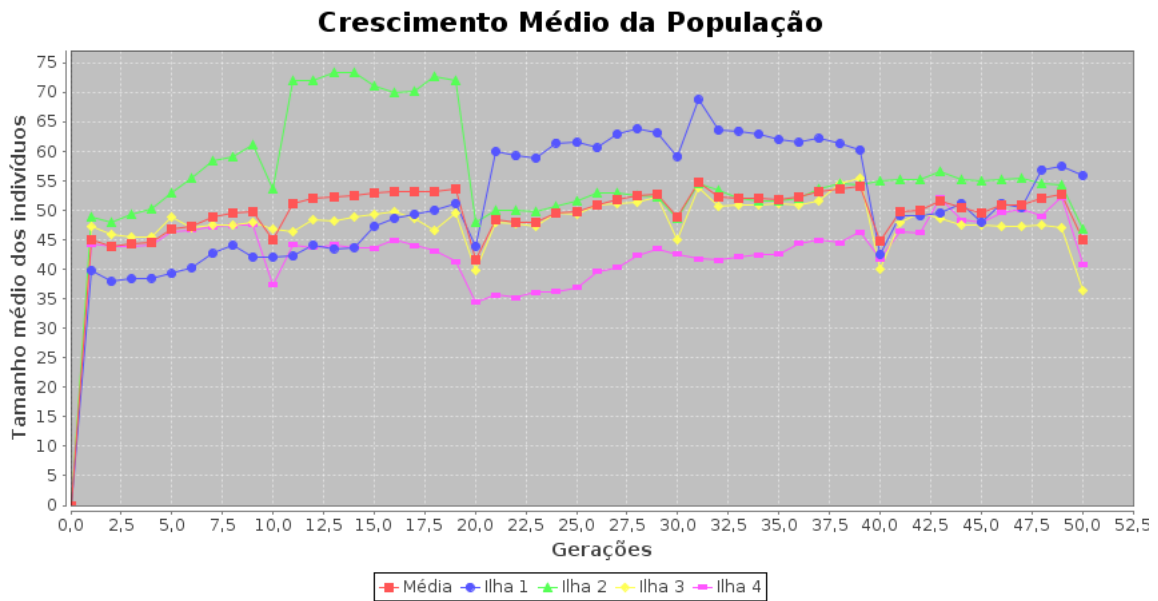


Figura 32 – Gráfico do tamanho médio da árvores.

com argumento (um inteiro que especifica a numeração da ilha) ou sem argumento (apresentando a informação global da população de todas as ilhas).

Para a confecção dos gráficos, foi utilizada a API JFreeChart (<http://www.jfree.org/jfreechart/>), uma interface de livre utilização. Esta API tem o inconveniente de trabalhar apenas com gráficos 2D.

As imagens, no formato PNG, são salvas no mesmo diretório onde se encontra a aplicação. Além disso, caso o aplicativo gnuplot (<http://www.gnuplot.info/>) esteja instalado na máquina em que a ferramenta é executada, um gráfico de comparação é gravado no mesmo diretório da aplicação.

5.1.6 Parâmetros padrão

A ferramenta criada possui configuração padrão, cujos valores de seus parâmetros foram obtidos por meio de revisão da literatura que englobou diversos trabalhos na área. A configuração adotada é:

Os valores *default* são:

- ☐ Taxa de cruzamento: 85%
- ☐ Taxa de mutação: 5%
- ☐ Taxa de reprodução: 10%
- ☐ Tamanho da população: 500
- ☐ Quantidade de gerações: 51

- ❑ Profundidade inicial das árvores: 6
- ❑ Tipo de cruzamento: *KOZA_CROSSOVER*
- ❑ Tipo de mutação: mutação de subárvore
- ❑ Tipo de função de aptidão: minimização do erro
- ❑ Inicialização da população: *ramped half-and-half*, com 90% de chances de escolha de nós função

5.1.7 Exemplo

O código a seguir utiliza a ferramenta para encontrar a adequada distribuição de portas lógicas para encontrar um circuito digital combinacional a partir de sua tabela verdade. A passagem do conjunto de amostras é feita por meio de duas matrizes e o programa é executado com os parâmetros *default*.

```
public class Teste {  
  
    public static void main(String[] args) {  
  
        ArrayList<IFuncao> portas = new ArrayList<>();  
        portas.add(new AND());  
        portas.add(new NAND());  
        portas.add(new NOR());  
        portas.add(new NOT());  
        portas.add(new OR());  
        portas.add(new XOR());  
  
        double[][] entradas = new double[][]{{1, 0, 0},  
                                                {1, 0, 1},  
                                                {1, 1, 0},  
                                                {1, 1, 1}};  
  
        double[][] saidas = new double[][]{{1, 0, 0, 0},  
                                              {0, 1, 0, 0},  
                                              {0, 0, 1, 0},  
                                              {0, 0, 0, 1}};  
  
        PG pg = new PG(entradas, saidas);  
  
        pg.addFuncao(portas);  
  
        pg.executaPG();  
  
        System.out.println(pg.getTempoUltimaExecucao());  
  
    }  
}
```

```
Nucelos disponíveis: 4
Tipo de execução: Paralela
Gerações em cada Ilha:
[0] = 51 ; [1] = 51 ; [2] = 51 ; [3] = 51 ;

Tempo de execução (ns): 3884057044

Erro: 0,000000
Aptidão: 1,035714
Funções: 16

Melhor indivíduo:
[0] -> (NOR (NAND (NOT z)(NOT y))(AND (NOR z x)(NAND z z)))
[1] -> (AND (NOT y)(NAND (NOT z)(NOT y)))
[2] -> (NOT (NAND (NOT z) y))
[3] -> (AND y z)
```

5.2 A ferramenta como aplicativo

Embora a ferramenta possa ser inserida diretamente em trechos de código de programas Java, essa pode não ser a abordagem preferida pela a maioria dos usuários. Uma interface gráfica que seja simples, intuitiva, mas suficientemente detalhada para se explorar recursos avançados da programação genética é preferível.

Esta seção apresenta o PPGP em sua versão “aplicativo”, que oferece todo o suporte necessário para usuário realizar regressões simbólicas.

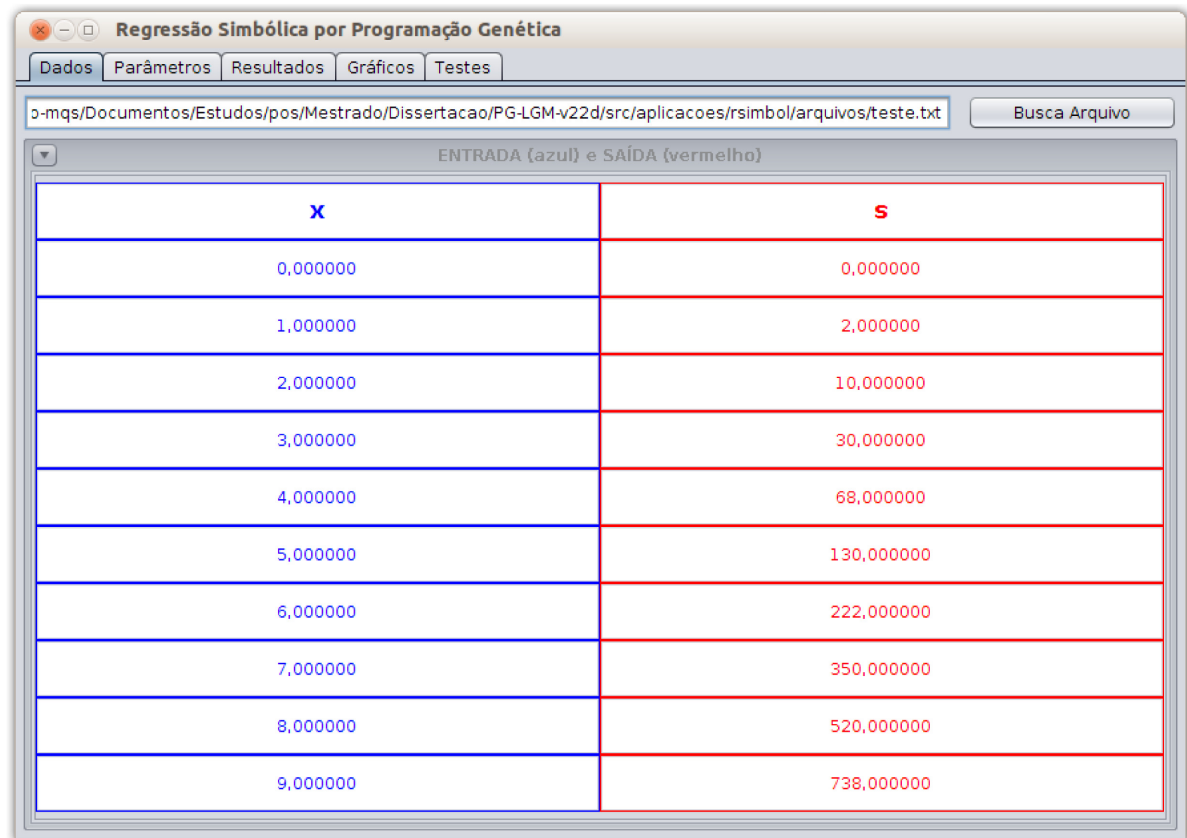
5.2.1 Aba Dados

Nesta aba, os dados correspondentes aos casos de teste são passadas à aplicação, que definem o padrão que se deseja mapear, conforme explicado na Seção 5.1.1. Isso é feito através do botão “Busca Arquivo”. Os casos de teste, conforme visto na Figura 33, são apresentados, com as entradas em azul e as saídas em vermelho.

5.2.2 Aba Parâmetros

A Figura 34 apresenta a aba de configuração de parâmetros e opções do programa. Quando executada, a aplicação traz uma configuração pré-estabelecida com os parâmetros *default*. Entretanto, o usuário pode modificar as taxas de cruzamento, mutação, tamanho da população, quantidade de gerações e profundidade das árvores na população inicial. O usuário pode também escolher entre executar a aplicação de forma paralela (padrão) ou sequencial, para efeito de testes e comparações.

Todas as funções disponíveis aparecem no painel “Funções”. Usuários com conhecimento em programação e que queiram adequar a aplicação acrescentando funções es-



X	S
0,000000	0,000000
1,000000	2,000000
2,000000	10,000000
3,000000	30,000000
4,000000	68,000000
5,000000	130,000000
6,000000	222,000000
7,000000	350,000000
8,000000	520,000000
9,000000	738,000000

Figura 33 – Buscar arquivo com padrões de treinamento.

pecíficas ao domínio do problema em análise podem fazê-lo conforme explicado na Seção 5.1.2.2. Para que a nova função apareça no painel Funções, é necessário modificar, na classe `RegressaoSimbolica`, o método `iniciaValores()`, acrescentando a linha de código `opDisp.add(new NovaFuncao());`.

```
private void iniciaValores() {
    opDisp = new ArrayList<>();

    opDisp.add(new Soma());
    opDisp.add(new Subtracao());
    ...
    opDisp.add(new NovaFuncao()); // cadastrando a nova funcao
    ...
}
```

Várias opções de cruzamento estão disponíveis no painel “Cruzamento”. A opção *default*, “*Crossover* ‘clássico’”, equivale ao *crossover* de sub-árvore proposto por Koza (1990a). As demais opções devem ser escolhidas de acordo com o problema em análise – a Seção 2.7.2 descreve cada um deles. Já o painel “Mutação” permite o uso dos vários tipos de mutação descritos na Seção 2.7.3.

As constantes podem ser aleatoriamente criadas antes do início da execução da programação genética e isso pode ser feito de duas maneiras: valores reais entre 0 e 1 –

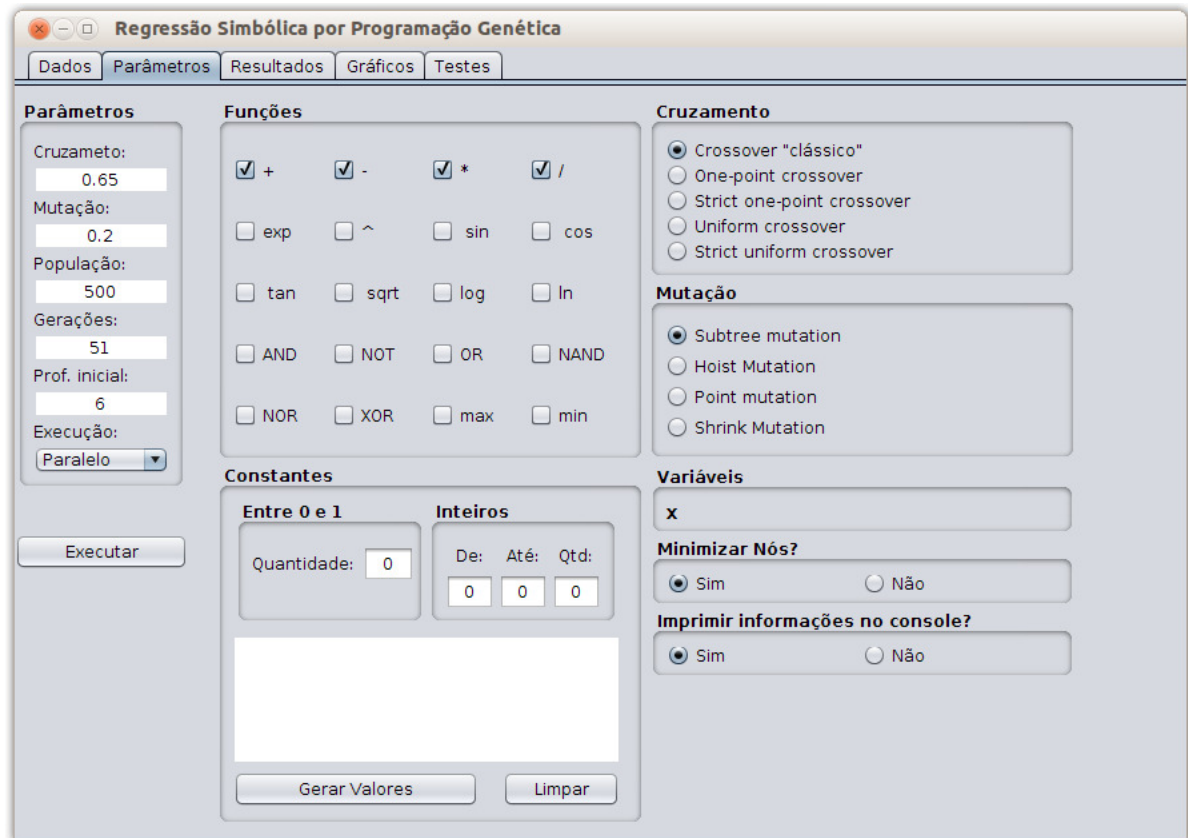


Figura 34 – Tela de configuração de parâmetros.

necessitando apenas indicar a quantidade delas – e valores inteiros dentro de um intervalo especificado, na quantidade desejada. Esta informação é opcional.

O painel Variáveis apenas mostra as variáveis utilizadas no problema em análise e sua quantidade é automaticamente determinada, de acordo com as características do conjunto de treinamento.

O painel “Minimiza nós?” indica a ação que será tomada no momento de escolha da melhor resposta dentro do conjunto Pareto-ótimo. Embora este conjunto contenha várias soluções não dominadas, todas válidas, pode-se optar por escolher aquela que tenha o menor erro.

5.2.3 Aba Resultados

A Figura 35 apresenta a aba de exibição dos resultados, que é automaticamente aberta logo após o término da execução da programação genética. Na caixa de texto são exibidas as informações da saída padrão da classe PG.

Para efeito de visualização, o botão “Árvore” apresenta a solução encontrada no formato de árvore.

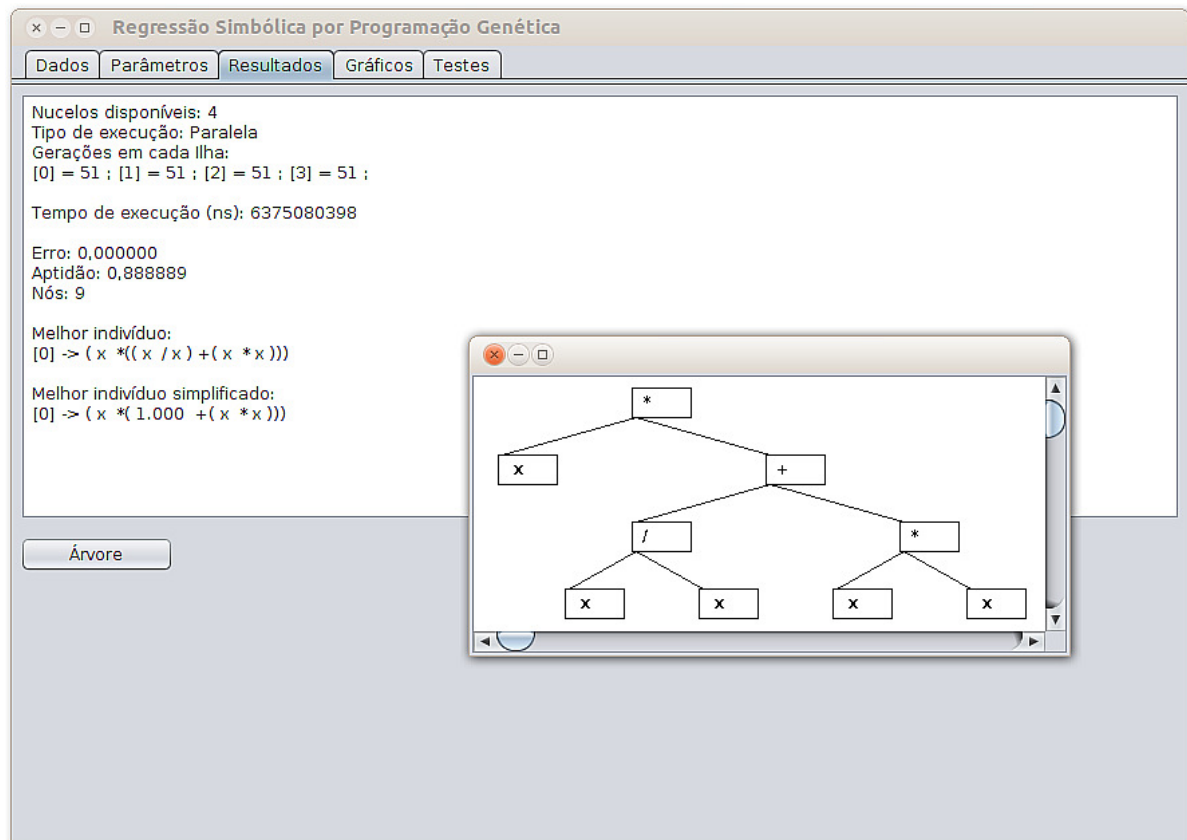


Figura 35 – Exibição dos resultados.

5.2.4 Aba Gráficos

Os gráficos são uma ferramenta de grande utilidade na análise de dados em GP. Por meio deles, é possível comparar a saída gerada pela equação encontrada e a distribuição dos pontos do conjunto de casos de teste pelo espaço de busca, ver crescimento do tamanho médio da população ao longo das gerações e a evolução da aptidão ao longo das gerações – tanto no que se refere ao melhor indivíduo em cada geração quanto à média das aptidões da população.

A Figura 36 apresenta um dos gráficos disponíveis na aplicação. Há a opção de visualizar a informação referente a uma ilha específica ou da população global. Já a Figura 37 apresenta o tamanho médio das árvores das subpopulações das ilhas, juntamente com a média de toda a população, ao longo das gerações.

5.3 Resultados e discussões

A ferramenta PPGP oferece suporte à utilização da programação genética por profissionais de diferentes níveis de conhecimento. As instruções contidas ao longo das primeiras seções conduzem o leitor à melhor utilização desta ferramenta.

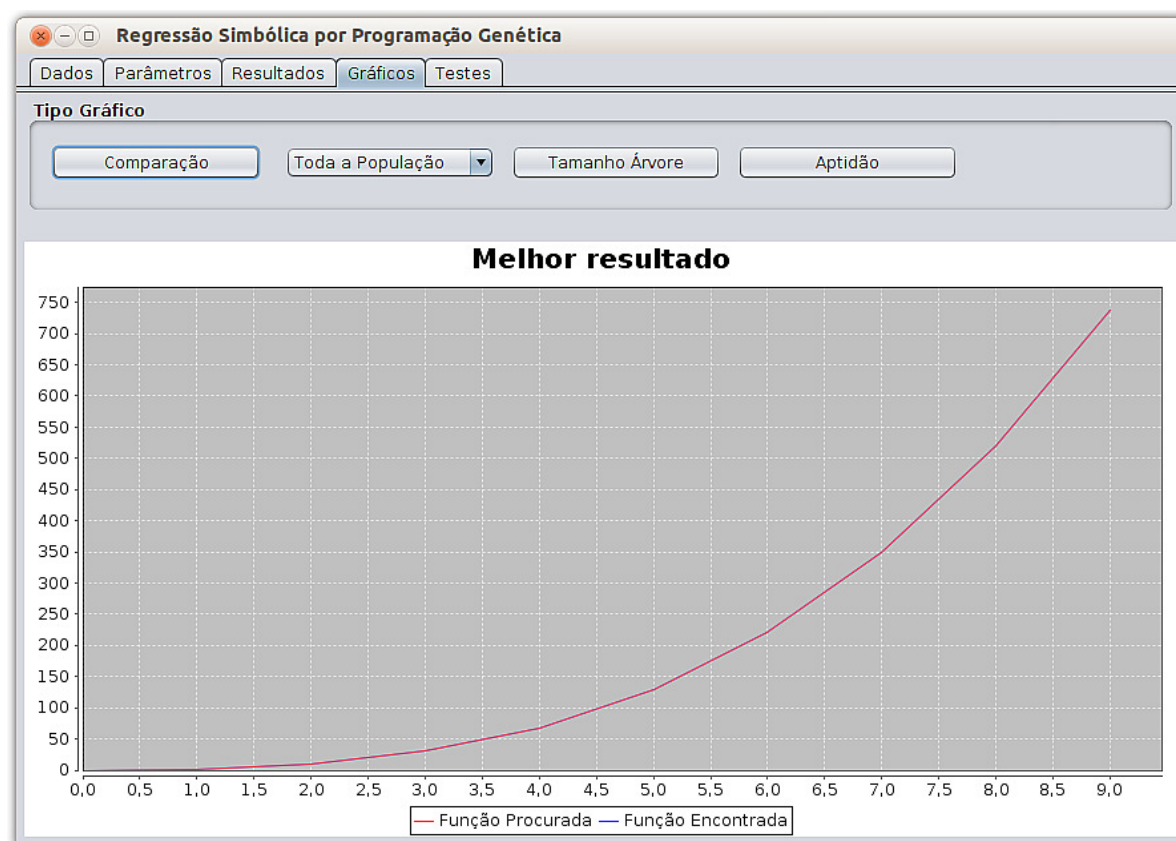


Figura 36 – Aba de exibição de gráficos. Em destaque, a comparação do melhor resultado encontrado, comparado com gráfico plotado pelo conjunto de casos de teste. Como o resultado foi exato, os gráficos se sobrepõem.

Além disso, o uso dos conceitos de dominância de Pareto permitiram a obtenção de respostas com estruturas menos complexas, o que pode ser evidenciado nos testes realizados, onde a saída corresponde (ou se aproxima bastante) da equação originalmente proposta. Durante as execuções, não se verificou o estouro de memória, significando, assim, que as médias do tamanho da população ao longo das gerações se mantiveram em níveis aceitáveis.

A eficiência, observada na velocidade de execução dos testes realizados, denotam as vantagens de se explorar o potencial dos processadores *multicore*.

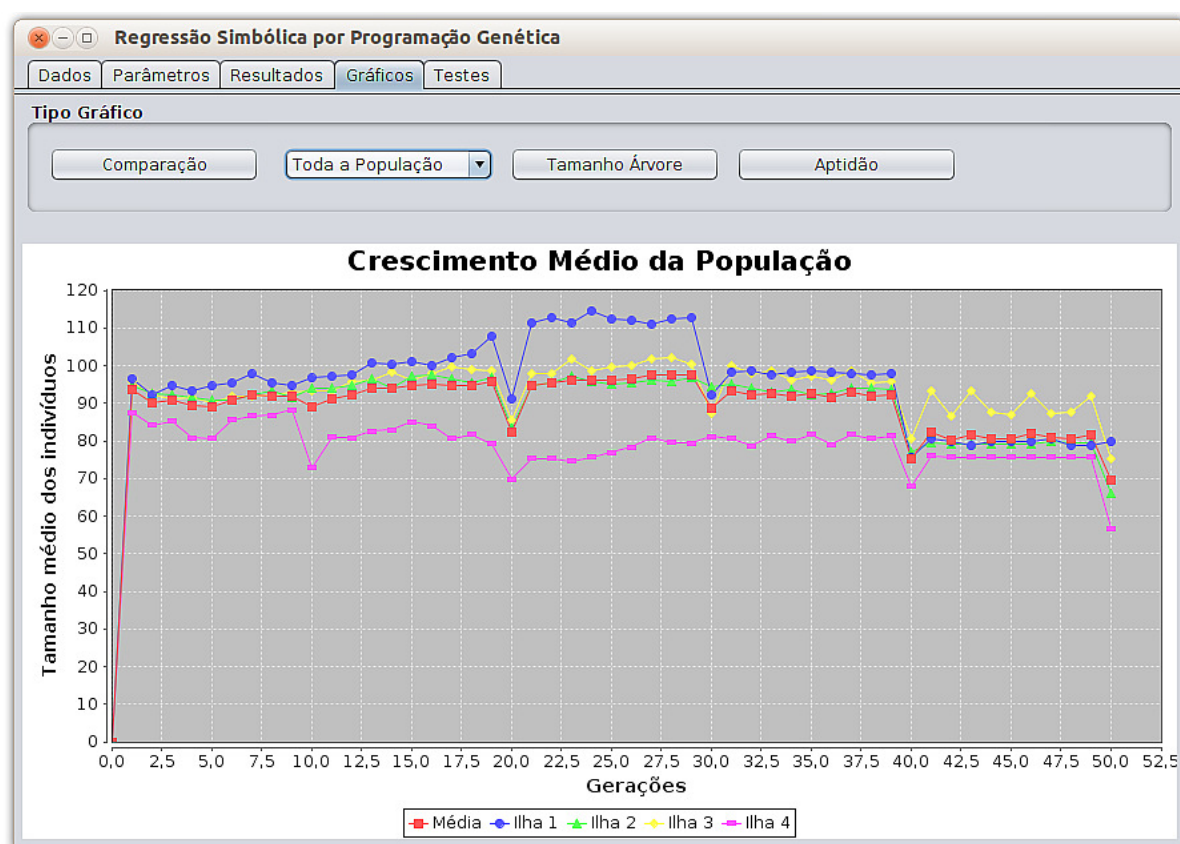


Figura 37 – Aba de exibição de gráficos. Em destaque, a o tamanho médio das árvores da população e das subpopulações ao longo das gerações.

Regressão simbólica via programação genética

A regressão simbólica consiste em encontrar uma expressão matemática (escrita simbolicamente) que seja capaz de gerar (de forma exata ou aproximada) um conjunto finito de resultados quando suas variáveis são submetidas a entradas particulares, fornecendo, assim, um modelo a partir de um conjunto de dados (KOZA, 1992b).

Por revelar equação formadora do conjunto de dados, a regressão simbólica se difere das demais técnicas de regressão. A regressão linear, por exemplo, determina os coeficientes da linha que melhor modela (com o menor erro quadrático médio) um conjunto de dados (LARSON; FARBER, 2010). De forma similar, a regressão quadrática determina os coeficientes que melhor modelam uma equação quadrática.

Neste capítulo, são apresentados diversos cenários em que a ferramenta PPGP é utilizada para obter equações formadoras de um conjunto de dados. Conforme feito por Grings (2006), optou-se por utilizar problemas de regressão simbólica mais simples (*toy problems*), sob a lógica de que o desempenho dos algoritmos se mantenha proporcional quando do uso de problemas mais complexos. Para todos os testes, utilizou-se os parâmetros padrão da ferramenta, conforme consta na Seção 5.1.6. As únicas diferenças ficam por conta das funções utilizadas como genes. Os experimentos foram realizados em um computador com processador Intel Core i3-2310M (2.10GHz), com 3,8 GB de memória física, com Sistema Operacional Ubuntu 13.04 64 bits.

Os resultados exibidos pelo programa tiveram suas operações protegidas devidamente tratadas, de forma que eles podem ser simplificados. Como o processo de simplificação simbólica de expressões matemáticas é um assunto complexo e acima do escopo deste trabalho, foi utilizado o *Maxima*, um software livre de álgebra computacional.

6.1 Funções polinomiais com uma variável

6.1.1 Polinômio de grau 2: $x^2 + x + 1$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , /
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 10229412927
Erro: 0,000310
Tamanho: 7
```

```
Resposta: [0] -> ( x +(( x * x ) + 1.000 ) )
```

6.1.2 Polinômio de grau 3: $x^3 + x^2 + x + 1$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , /
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 7060633915
Erro: 0,001893
Tamanho: 27
```

```
Resposta: [0] -> ((( x + 0.000 ) + 1.000 ) -((( 2.000 * x ) *((
    x * x ) *( 1.000 -( x * x )))))/( ( 2.000 * x ) *( x -
    1.000 ))))
```

6.1.3 Polinômio de grau 4: $x^4 + x^3 + x^2 + x$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , /
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 7268453874
Erro: 0,041074
Tamanho: 37
```

```
Resposta: [0] -> (( 2.000 * x ) +(((( x * x ) - 1.000 ) *((
    2.000 * x ) -( x -( x * x )))))+( (( x * 1.000 ) -( 2.000 *
    x )) -(( 0.000 - x ) -( 2.000 *( x * x ))))))
```

6.2 Funções polinomiais com duas variáveis

6.2.1 Polinômio $x^2 + y^2 + 1$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , /
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 8912795704
Erro: 0,000523
Tamanho: 15
```

```
Resposta: [0] -> ((( x * x ) +(( 0.000 -( y - 1.000 )) +( y *
y ))) + y )
```

6.3 Funções com logaritmos

6.3.1 Função $5 * \log(x) + x + 1$

```
Nucelos disponíveis: 4 (execução paralela)
Funções: + , - , * , / , log
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 5622006375
Erro: 0,481159
Tamanho: 25
```

```
Resposta: [0] -> (( x / 1.000 ) +(log ((((( x * x ) - 1.000 )
/(log (( 2.000 * x )))))+( x * x )*(( x * x ) * x ))))
```

6.4 Funções trigonométricas com uma variável

6.4.1 Função $(\sin x)/x$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , / , sin , cos
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 30222657273
Erro: 0,000030
Tamanho: 8
```

```
Resposta: [0] -> ( x *(( sin ( x )) /( x * x )))
```

6.4.2 Função $\sin x + \cos x$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , / , sin , cos
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 16992446291
Erro: 0,000041
Tamanho: 5
```

```
Resposta: [0] -> (( cos ( x )) +( sin ( x )))
```

6.4.3 Função $\sin^2 x + \cos x$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , / , sin , cos
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 37340095492
Erro: 0,850549
Tamanho: 15
```

```
Resposta: [0] -> (( cos ((( cos ( x )) / ( sin (( cos (( cos
  (( cos (( cos (( cos (( cos ( x ))))))))))))))) + ( cos (
    x )))
```

6.5 Funções trigonométricas com duas variáveis

6.5.1 Função $\sin^2 x + \cos y$

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , / , sin , cos
Constantes: geradas ao longo das gerações.
Tempo de execução (ns): 26608714014
Erro: 0,000042
Tamanho: 8
```

```
Resposta: [0] -> ((( sin ( x )) *( sin ( x ))) + ( cos ( y )))
```

6.5.2 Observações

Nota-se que a obtenção do polinômio de grau 2 demandou mais tempo que as de graus superiores. Isso se deve, principalmente, ao tamanho dos indivíduos presentes na população inicial, que é formada, em sua maioria, por equações de comprimento maior que a resposta desejada – um polinômio de grau 2 tende ter estrutura mais simples do que outro de grau 4, por exemplo. Por essa razão, a evolução da população deve conduzir o aparecimento de indivíduos de comprimento menor do que seus antecessores, contrariando a tendência natural de crescimento do tamanho médio dos indivíduos da população – efeito *bloat*.

6.6 Reescrevendo equações

É possível reescrever equações ou obter razoáveis aproximações de equações conhecidas escolhendo um conjunto diferente de funções. Por exemplo, a função *Kotanchek*, dada pela Equação 15, pode ser reescrita sem o uso da função exponencial, trabalhando apenas com as operações +, -, * e /, como mostrado na Equação 16.

$$f(x, y) = \frac{e^{-(y-1)^2}}{1.2 + (x - 2.5)^2} \quad (15)$$

$$\frac{y}{y \left(\frac{1}{xy((x-y)^2 y - xy + x^2 + x)} \right) + x^2 (x^2 - 1.0 y) + \frac{0.5}{x}} + y^2 + 1.0 y + x \quad (16)$$

Neste caso, a saída do programa é:

```
Núcleos disponíveis: 4 (execução paralela)
Funções: + , - , * , /
Constantes: geradas ao longo das gerações.
```

Tempo de execução (ns): 43578450909
 Erro: 1,516383
 Tamanho: 61

Resposta: [0] -> (y /((((2.000 * y) +(x - y)) +((((y /
 x) / y) /((((x - y) *((x - y) * y)) +(x -(y * x
)) +(x * x))) * y)) +((1.000 /(2.000 * x)) +(x * x
) *((x * x) -(y / 1.000))))) * y)) +(y * y)))

6.7 Síntese de circuitos digitais combinacionais

A ferramenta PPGP pode ser também utilizada para realizar a síntese de circuitos digitais combinacionais. Tais circuitos são compostos por portas lógicas e seu comportamento pode ser descrito por tabelas-verdade.

Um multiplexador com 6 entradas, conforme descrito em Koza (1990b), foi modelado pela PPGP de acordo com o apresentado na Figura 38. Como pode ser visto abaixo, na saída do aplicativo, a resposta gera os valores exatos repassados pela tabela verdade:

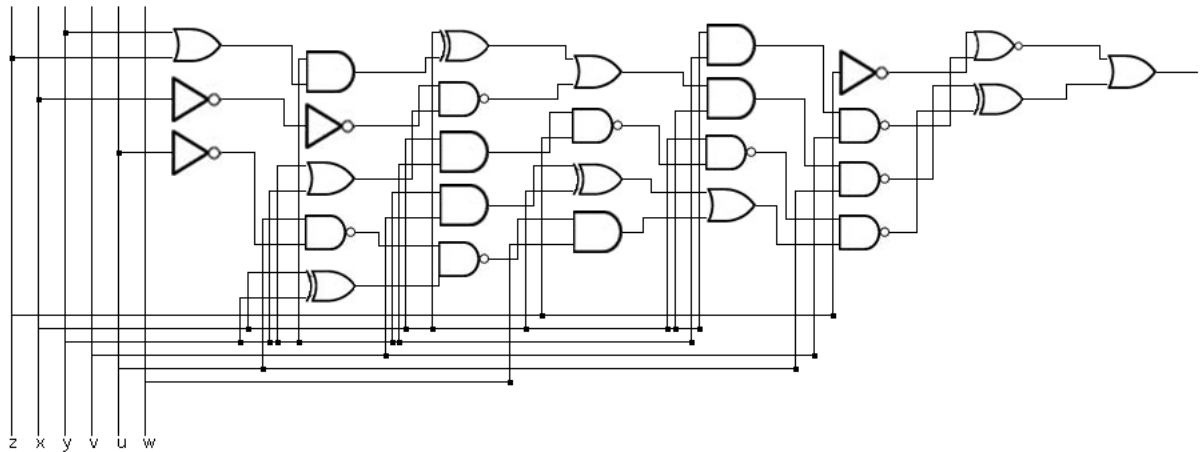


Figura 38 – Multiplexador de 6 entradas modelado pela ferramenta PPGP.

Núcleos disponíveis: 4 (execução paralela)
 Funções: AND , NAND , NOR , NOT , OR , XOR
 Constantes: geradas ao longo das gerações.
 Tempo de execução (ns): 2276660593
 Erro: 0,000000
 Tamanho: 53

Resposta: [0] -> (((NOT (z)) NOR((x AND y) NAND v)) OR((((
 x XOR(y AND(y OR z))) OR((NOT ((NOT (x))) NAND(y OR
 y))) AND x) NAND u) XOR((x NAND((x AND y) NAND z))
 NAND(((y AND v) XOR x) OR(((u NAND(NOT (u))) NAND(x
 XOR y)) AND w))))))

Os multiplexadores possuem uma entrada e várias saídas. Pode-se também obter a configuração de demultiplexadores como o de 1 para 8 linhas, que possui 3 entradas (dadas por x, y, z) e 8 saídas, numeradas de 0 a 7:

Núcleos disponíveis: 4 (execução paralela)
 Funções: AND , NAND , NOR , NOT , OR , XOR
 Constantes: geradas ao longo das gerações.
 Tempo de execução (ns): 35418155327
 Erro: 0,000000
 Tamanho: 167

Resposta:

```
[0] -> ((( y XOR z ) XOR( z OR z )) NOR(( z NOR x ) NAND( y
    NAND y )))
[1] -> (((NOT (((NOT ( x )) OR(NOT ( z ))))) OR y ) NOR( z NOR((
    NOT ( x )) XOR(( x NAND x ) XOR(NOT ((NOT (( z AND x ))))))))
    )
[2] -> ((( z NOR z ) NAND y ) NOR( z XOR x ))
[3] -> ((NOT ((NOT (( x XOR z ))))) XOR((( z AND(( x NOR y )
    NOR((NOT (( z AND z )) OR z ))) XOR( y NOR z )) XOR(( x XOR
    y ) NOR(( y AND z ) AND y ))))
[4] -> ((((( z AND( z OR(NOT (( x AND(NOT (( z OR y ))))))))
    NAND((NOT (( x NOR z )) NOR z )) AND(( z NOR y ) AND x )) OR
    (( z NOR(NOT (( x XOR z )))) AND((NOT ( x )) XOR(( z NOR z )
    AND(( z AND y ) OR(NOT ((NOT (( x AND z )) AND(NOT ( z ))
    ))))))))
[5] -> ((( z OR x ) AND( z AND( y XOR x ))) AND x )
[6] -> ((( y AND( z NOR z )) AND y ) AND(( y AND z ) OR( x
    AND x )))
[7] -> ((NOT ( x )) NOR(( y NAND y ) OR(NOT (( y AND z )))))
```

6.8 Modelagem do lançamento oblíquo no vácuo

A modelagem de sistemas físicos está entre as principais utilizações da programação genética. A extração das expressões matemáticas que geram um conjunto de dados é de grande interesse prático e teórico. Nesta Seção, será explorado um problema simples de física para ilustrar o potencial da regressão simbólica via programação genética.

O problema aqui tratado é o lançamento oblíquo, que consiste em encontrar e equação da parábola gerada pelo lançamento de um corpo e um ângulo θ em relação ao solo. Esta equação é dada pela relação entre a altura alcançada (S), que varia em função do tempo t . Considera-se também a velocidade inicial de lançamento v_0 . Convenciona-se que o sistema localiza-se no vácuo, tendo como única influência a aceleração da gravidade g . Todos esses valores são regidos pela Equação 17.

$$S = v_0 t - \frac{gt^2}{2} \quad (17)$$

Para o estudo de caso, considerou-se $v_0 = 25m/s$, e $g = 10m/s^2$. Assim, a Equação 17 é reescrita na forma da Equação 18. Foram gerados 50 pares (S, t) , com $t \in [0, 5]$, que serviram de entrada para a aplicação PPGP, que foi executada com os parâmetros *default*.

$$S = 25t - 5t^2 \quad (18)$$

Uma aproximação encontrada foi:

```
( x +(( 2.000 * x ) +(((( 2.000 * x ) +(( x + 1.000 ) +
( x -((( x + ( ( x * x ) -( 2.000 * x ))) +(( 2.000 * x ) +
(( 2.000 * ( x * x )) - (( 2.000 * x ) + x )))) + ( x -
( 2.000 * x )))))) +((( 2.000 * x ) + ( ( 2.000 * x ) +
(( x + 0.000 ) +( x -( x * x ))))) + x )) + ((( 1.000 +
( 2.000 * x )) +(( 2.000 * x ) +( x -( x * x )))) +
( 2.000 * x )))))).
```

Simplificada, esta expressão corresponde à Equação 19 que, como pode ser visto na Figura 39, gera valores bem próximos aos ideais.

$$2 + 24x - 5x^2 \quad (19)$$

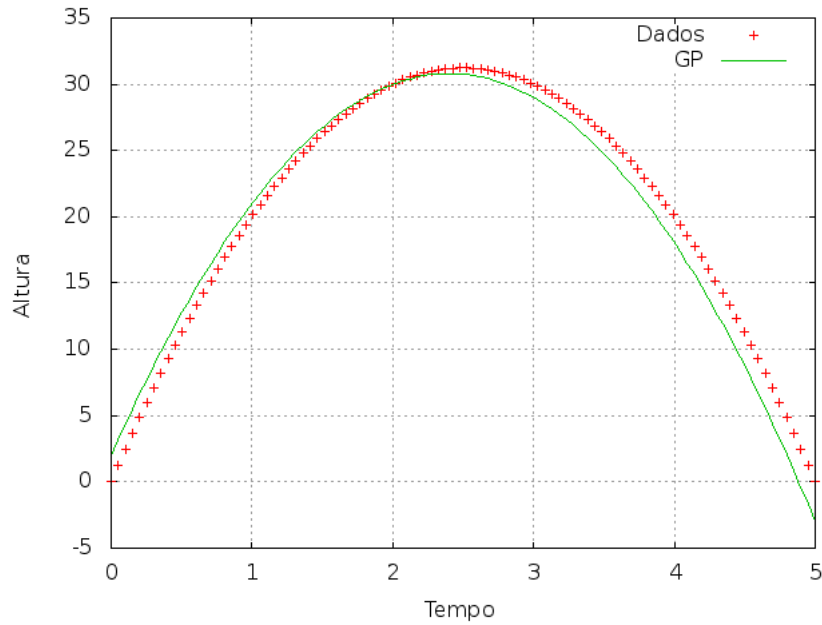


Figura 39 – Aproximação do lançamento oblíquo.

Em outra execução, encontrou-se o valor exato, dado pela expressão:

```
((((( ((( ( 2.000 * x ) / ( x / ( 2.000 * x )) ) + ( x * x )) +
( x +(( x -( x * x )) -( x * x )))) +(((( ( x +( 2.000 *
( 2.000 * x )) ) / ( x /((( ( x -( ( x * x ) + x )) +
( x * x )) + ( x +(( x -( x * x )) -( ( x + ( ( x -( x * x )) -
(( x +( x -( x * x )) ) - x )) ) - x )))) + x )) + x ) +
( x * x )) + ( x + -1.000 )) + x )) + x ) + 1.000 ).
```

Sua simplificação resulta na Equação 18 e seu gráfico é apresentado na Figura 40.

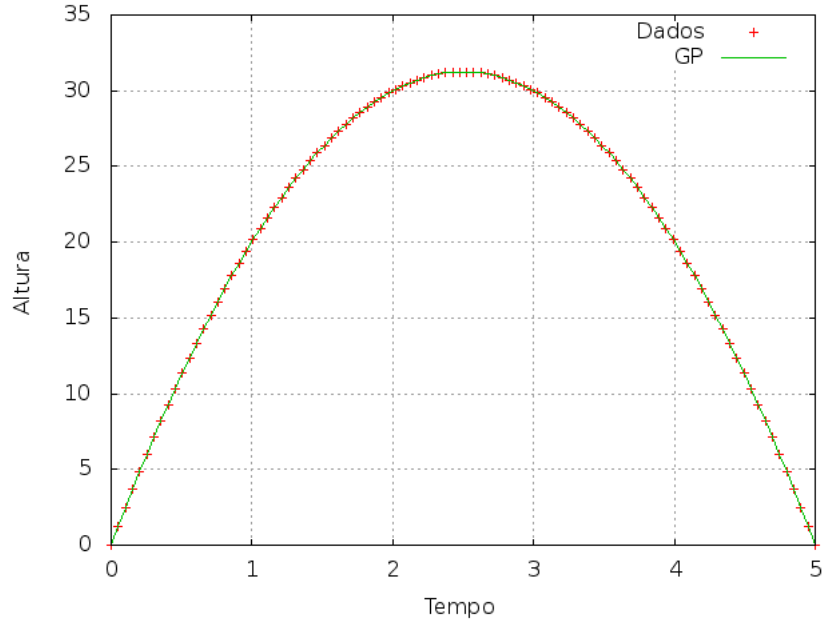


Figura 40 – Lançamento oblíquo.

6.9 Aproximação da função dupla exponencial para medir descargas elétricas atmosféricas

O efeito da tensão induzida por descargas atmosféricas em linhas de transmissão elétrica não é assunto recente para a Engenharia Elétrica. Trabalhos como o de Anderson e J. (1980) tiveram o intuito de medir e caracterizar este tipo de tensão. Porém, o mais comum é a utilização da função dupla exponencial, que modela a tensão da descarga elétrica em função do tempo. Esta função é dada pela Equação 20 (NUCCI et al., 1993).

$$I(t) = \frac{I_0}{\eta} \frac{(t/\tau_1)^n}{1 + (t/\tau_1)^n} e^{(-t/\tau_2)}, \text{ com } \eta = e^{(-(\tau_1/\tau_2)(n\tau_2/\tau_1)^{(1/n)})}. \quad (20)$$

onde:

- I_0 : Amplitude da corrente na base do canal;
- τ_1 : Constante relacionada ao tempo de frente da onda de corrente;
- τ_2 : Constante relacionada ao tempo de decaimento da onda de corrente;
- η : Fator de correção de amplitude;
- n : Expoente (2 a 10).

Nesta seção, utiliza-se a programação genética para obter uma equação que modele a tensão em função do tempo e que seja uma alternativa à dupla exponencial. Para isso, um conjunto de 100 valores distribuídos no intervalo $[0, 10]$ foi gerado a partir da Equação 20.

Os valores atribuídos aos parâmetro da Equação 20 foram obtidos no trabalho de Campos (2012) e são apresentadas na Tabela 5.

Tabela 5 – Valores de parâmetro para a Equação 20.

$I_0(kA)$	$\tau_1(\mu s)$	$\tau_2(\mu s)$	n
15,4	0,6	4,0	3,4

A exemplo do que foi feito na Seção 6.6, optou-se por modelar a função somente com as quatro operações básicas (+, −, *, /), obtendo, assim, uma expressão sem exponencial. O melhor resultado aproximado é exibido na Equação 21.

$$\frac{0.49 - x}{\frac{0.874}{\frac{2.0x}{x+0.742} + x - 0.062} + \frac{2.0x}{x+0.326} + 0.049} + \frac{106.435 (x + 0.049)}{(x + 0.472) \left(x + \frac{0.953172}{x^2}\right)} + 0.683 \quad (21)$$

A Figura 41 apresenta os gráficos gerados pela função original, que forneceu os pontos para a programação genética, e a curva gerada pela Equação 21

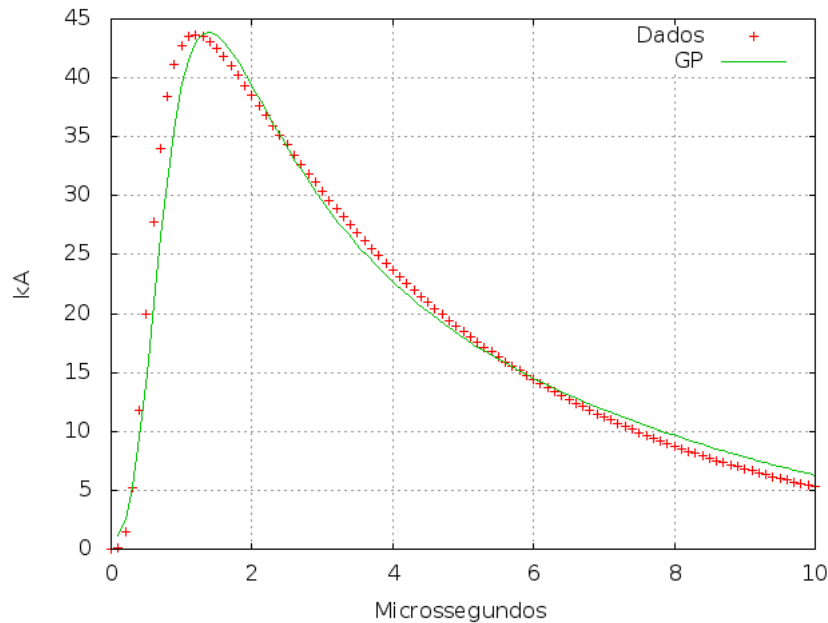


Figura 41 – Aproximação da função dupla exponencial.

Conclusão

O ocultamento da complexa manipulação das estruturas de dados subjacente e a possibilidade de adicionar funções personalizadas faz da PPGP, mesmo quando utilizada na forma de API, útil o bastante para ser utilizada em variados campos de aplicação. Por facilitar o uso da programação genética, considera-se que o objetivo principal deste trabalho foi alcançado.

Os objetivos secundários, necessários ao melhor funcionamento da ferramenta, também foram atingidos. Com a redução do efeito *bloat*, as saídas produzidas pela GP mostraram-se mais simples e compactas, mas sem prejudicar o valor de aptidão dos indivíduos que formam a população ao longo das gerações. Já no que se refere ao desempenho, o uso do *framework fork/join* permitiu a implementação do modelo de ilhas em programação genética aproveitando o potencial dos processadores *multicore*. O tempo de execução na versão paralela apresentou diminuição proporcional ao número de núcleos físicos presentes na máquina.

Os problemas de regressão simbólica apresentados no Capítulo 6, resolvidos via programação genética, permitem vislumbrar diversas possibilidades de aplicação. A modelagem da função dupla exponencial, vista na Seção 6.9, é um exemplo de modelagem de um problema físico de grande importância prática.

Trabalhos futuros

Processamento em GPU

A melhoria de desempenho obtida por meio da paralelização em processadores *multicore* mostrou que é possível ter um sistema de programação genética capaz de trabalhar com grandes populações utilizando um hardware de baixo custo.

Além dos processadores *multicore*, os avanços recentes na tecnologia de *Graphics Processing Unit* (GPU) faz com que esse tipo de equipamento torne-se também atrativo, por se tratar de hardware de baixo custo e por fornecer potencial de processamento superior

aos processadores convencionais. Maitre (2011) faz uso de programação genética neste tipo de placas de processamento gráfico para o tratamento de problemas reais de modelagem. Porém, sua adequada manipulação requer avançadas habilidades de programação.

O próximo passo é ampliar as funcionalidades da ferramenta PPGP, estendendo o uso da programação genética paralela com Pareto para placas GPU.

Uso de gramáticas

Gramáticas podem também ser utilizadas em sistemas de programação genética. Essa abordagem permite maior liberdade de representação, por oferecer meios que garantem, pelo conceito de árvores de derivação, a produção de programas válidos ao longo da evolução.

Outro passo para ampliação da utilidade da ferramenta desenvolvida neste trabalho é permitir o uso de gramáticas.

Outras funcionalidades

Outros recursos que podem trazer benefícios aos usuários da PPGP é a inclusão de representações mais generalizadas dos indivíduos, como grafos e listas.

O uso de ferramentas mais robustas para a exibição dos gráficos em três dimensões também faz-se necessárias.

Referências

AKHTER, S.; ROBERTS, J. **Multi-Core Programming**. first edition. [S.l.]: Intel Corporation, 2006. 336 p.

ANDERSON, R. B.; J., E. A. Lightning parameters for engeneering applications. **Electra**, 1980. 1980.

ANDRE, D.; KOZA, J. R. Parallel genetic programming: A scalable implementation using the transputer network architecture. In: ANGELINE, P. J.; Kinnear, Jr., K. E. (Ed.). **Advances in Genetic Programming 2**. Cambridge, MA, USA: MIT Press, 1996. cap. 16, p. 317–338. ISBN 0-262-01158-1. Disponível em: <<http://cisnet.mit.edu/Advances-in-Genetic-Programming/334>>.

ANGELINE, P. J. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In: KOZA, J. R. et al. (Ed.). **Genetic Programming 1996: Proceedings of the First Annual Conference**. Stanford University, CA, USA: MIT Press, 1996. p. 21–29. Disponível em: <<http://www.natural-selection.com/Library/1996/gp96.zip>>.

ARAÚJO, S. G. de. **Síntese de sistemas digitais utilizando técnicas evolutivas**. Tese (Doutorado) — Programa de Pós-Graduação de Engenharia da Universidade Federal do Rio de Janeiro, 2004.

BANZHAF, W. Genetic programming for pedestrians. In: FORREST, S. (Ed.). **Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93**. University of Illinois at Urbana-Champaign: Morgan Kaufmann, 1993. p. 628. Disponível em: <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z>.

BLEULER, S. et al. Multiobjective genetic programming: Reducing bloat using SPEA2. In: **Proceedings of the 2001 Congress on Evolutionary Computation CEC2001**. COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea: IEEE Press, 2001. p. 536–543. ISBN 0-7803-6658-1. Disponível em: <<http://citeseer.ist.psu.edu/443099.html>>.

BLICKLE, T.; THIELE, L. **A Comparison of Selection Schemes Used in Genetic Algorithms**. Gloristrasse 35, 8092 Zurich, Switzerland, 1995. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2449>>.

BÄCK, T.; HAMMEL, U.; SCHWEFEL, H.-P. Evolutionary computation: Comments on the history and current state. **IEEE Transactions on Evolutionary Computation**, 1997. v. 1, p. 3–17, 1997.

CAMPOS, A. F. M. de. **Cálculo de sobretensões causadas por descargas atmosféricas indiretas em linhas de distribuição aéreas considerando falhas de isolamento**. Dissertação (Mestrado) — Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, 2012.

CHELLAPILLA, K. Evolutionary programming with tree mutations: Evolving computer programs without crossover. In: KOZA, J. R. et al. (Ed.). **Genetic Programming 1997: Proceedings of the Second Annual Conference**. Stanford University, CA, USA: Morgan Kaufmann, 1997. p. 431–438.

CORMEN, T. H. et al. **Introduction to Algorithms**. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0070131511.

CRAMER, N. L. A representation for the adaptive generation of simple sequential programs. In: **Proceedings of the 1st International Conference on Genetic Algorithms**. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985. p. 183–187. ISBN 0-8058-0426-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=645511.657085>>.

DEB, K. et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. **Evolutionary Computation, IEEE Transactions on**, 2002. v. 6, n. 2, p. 182–197, 2002. ISSN 1089-778X.

DEITEL, P.; DEITEL, H. **Java como programar**. 8. ed. São Paulo – SP, Brasil: Pearson Prentice Hall, 2009. ISBN 978-85-7605-194-7.

D’HAESELEER, P. Context preserving crossover in genetic programming. In: **Proceedings of the 1994 IEEE World Congress on Computational Intelligence**. Orlando, Florida, USA: IEEE Press, 1994. v. 1, p. 256–261. Disponível em: <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/WCCI94_CPC-.ps.Z>.

DUMITRESCU, D. et al. **Evolutionary computation**. Boca Raton, FL, USA: CRC Press, Inc., 2000. ISBN 0-8493-0588-8.

FAUSETT, L. (Ed.). **Fundamentals of neural networks: architectures, algorithms, and applications**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN 0-13-334186-0.

FOGEL, L. J. Autonomous automata. **Industrial Research**, 1962. v. 4, p. 14–19, 1962.

FONSECA, C. M.; FLEMING, P. J. An overview of evolutionary algorithms in multiobjective optimization. **Evol. Comput.**, 1995. MIT Press, Cambridge, MA, USA, v. 3, n. 1, p. 1–16, mar. 1995. ISSN 1063-6560. Disponível em: <<http://dx.doi.org/10.1162/evco.1995.3.1.1>>.

FOULDS, L. R. **Graph Theory Applications**. Springer, 1991. 203–207 p. Hardcover. ISBN 0387975993. Disponível em: <<http://www.amazon.com/gp/product/8173190461>>.

- FUJIKO, C.; DICKINSON, J. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In: **Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application**. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987. p. 236–240. ISBN 0-8058-0158-8. Disponível em: <<http://dl.acm.org/citation.cfm?id=42512.42544>>.
- GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675.
- GOODRICH, M. T.; TAMASSIA, R. **Data structures and algorithms in Java (3. ed.)**. [S.l.]: Wiley, 2003. I-XVII, 1-681 p. ISBN 978-0-471-64452-1.
- GRAHAM, P. **ANSI Common Lisp**. Upper Saddle River, NJ, USA: Prentice Hall Press, 1996. ISBN 0-13-370875-6.
- GRINGS, A. **Regressão Simbólica via Programação Genética: um Estudo de Caso com Modelagem Geofísica**. Dissertação (Mestrado) — Faculdade de Computação da Universidade Federal de Uberlândia, 2006.
- HARRIES, K.; SMITH, P. Exploring alternative operators and search strategies in genetic programming. In: KOZA, J. R. et al. (Ed.). **Genetic Programming 1997: Proceedings of the Second Annual Conference**. Stanford University, CA, USA: Morgan Kaufmann, 1997. p. 147–155. Disponível em: <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/harries.gp97_paper.ps.gz>.
- HICKLIN, J. **Application of the Genetic Algorithm to Automatic Program Generation**. University of Idaho, 1986. Disponível em: <<http://books.google.com.br/books?id=e87jtgAACAAJ>>.
- HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. Ann Arbor, MI, USA: University of Michigan Press, 1975.
- KINNEAR, J. K. E. Fitness landscapes and difficulty in genetic programming. In: **Proceedings of the 1994 IEEE World Conference on Computational Intelligence**. Orlando, Florida, USA: IEEE Press, 1994. v. 1, p. 142–147. ISBN 0-7803-1899-4. Disponível em: <<http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.wcci.ps.Z>>.
- KOZA, J. R. **Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems**. Stanford, CA, USA, 1990.
- _____. A hierarchical approach to learning the boolean multiplexer function. In: RAWLINS, G. J. E. (Ed.). **FOGA**. [S.l.]: Morgan Kaufmann, 1990. p. 171–192. ISBN 1-55860-170-8.
- _____. Genetic evolution and co-evolution of computer programs. In: LANGTON, C. G. et al. (Ed.). **Artificial Life II**. [S.l.]: Addison Wesley Publishing Company, 1992. p. 313–324.
- _____. **Genetic Programming: On the Programming of Computers by Means of Natural Selection**. Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-11170-5.

_____. **Genetic programming II: automatic discovery of reusable programs.** Cambridge, MA, USA: MIT Press, 1994. ISBN 0-262-11189-6.

_____. Advances in evolutionary computing. In: GHOSH, A.; TSUTSUI, S. (Ed.). New York, NY, USA: Springer-Verlag New York, Inc., 2003. cap. Human-competitive applications of genetic programming, p. 663–682. ISBN 3-540-43330-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=903758.903785>>.

LANGDON, W. B. The evolution of size in variable length representations. In: **1998 IEEE International Conference on Evolutionary Computation.** Anchorage, Alaska, USA: IEEE Press, 1998. p. 633–638. ISBN 0-7803-4869-9. Disponível em: <http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.wcci98_bloat.pdf>.

_____. Size fair and homologous tree genetic programming crossovers. **Genetic Programming and Evolvable Machines**, 2000. v. 1, n. 1/2, p. 95–119, apr 2000. ISSN 1389-2576. Disponível em: <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL_fairxo.pdf>.

LANGDON, W. B. et al. **The Evolution of Size and Shape.** Cambridge, MA, USA: MIT Press, June 1999. 163–190 p. Disponível em: <<http://www.cs.bham.ac.uk/~wbl/aigp3/ch08.ps.gz>>.

LARSON, R.; FARBER, B. **Estatística aplicada.** Quarta edição. [S.l.]: Pearson Prentice Hall, 2010.

LINDEN, R. (Ed.). **Algoritmos Genéticos: uma importante ferramenta da Inteligência Computacional.** Rio de Janeiro, RJ, Brasil: Brasport, 2008. ISBN 978-85-7452-373-6.

LUKE, S. **Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.** Tese (Doutorado) — Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA, 2000. Disponível em: <<http://www.cs.gmu.edu/~sean/papers/thesis2p.pdf>>.

LUKE, S. **Two Fast Tree-creation Algorithms for Genetic Programming.** Piscataway, NJ, USA: IEEE Press, set. 2000. 274–283 p. Disponível em: <<http://dx.doi.org/10.1109/4235.873237>>.

_____. Modification point depth and genome growth in genetic programming. **Evolutionary Computation**, 2003. v. 11, n. 1, p. 67–106, Spring 2003.

LUKE, S.; PANAIT, L. A comparison of bloat control methods for genetic programming. **Evolutionary Computation**, 2006. v. 14, n. 3, p. 309–344, Fall 2006. ISSN 1063-6560.

LUKE, S.; SPECTOR, L. A comparison of crossover and mutation in genetic programming. In: KOZA, J. R. et al. (Ed.). **Genetic Programming 1997: Proceedings of the Second Annual Conference.** Stanford University, CA, USA: Morgan Kaufmann, 1997. p. 240–248. Disponível em: <<http://www.cs.gmu.edu/~sean/papers/comparison/comparison.pdf>>.

MAITRE, O. **GPGPU for Evolutionary Algorithms.** Tese (Doutorado) — Université de Strasbourg, 2011.

MAXWELL, S. R. Why might some problems be difficult for genetic programming to find solutions? In: KOZA, J. R. (Ed.). **Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996**. Stanford University, CA, USA: Stanford Bookstore, 1996. p. 125–128. ISBN 0-18-201031-7.

MCKAY, B.; WILLIS, M. J.; BARTON, G. W. Using a tree structured genetic algorithm to perform symbolic regression. In: ZALZALA, A. M. S. (Ed.). **First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications**, GALEZIA. Sheffield, UK: IEE, 1995. v. 414, p. 487–492. ISBN 0-85296-650-4.

NUCCI, C. et al. Lightning-induced voltages on overhead lines. **Electromagnetic Compatibility, IEEE Transactions on**, 1993. v. 35, n. 1, p. 75–86, 1993. ISSN 0018-9375.

O'REILLY, U.-M.; OPPACHER, F. The troubling aspects of a building block hypothesis for genetic programming. In: WHITLEY, L. D.; VOSE, M. D. (Ed.). **Foundations of Genetic Algorithms 3**. Estes Park, Colorado, USA: Morgan Kaufmann, 1994. p. 73–88. ISBN 1-55860-356-5. Published 1995. Disponível em: <<http://citeseer.ist.psu.edu/oreilly92troubling.html>>.

OUSSAIDÈNE, M. et al. Parallel genetic programming: an application to trading models evolution. In: **Proceedings of the First Annual Conference on Genetic Programming**. Cambridge, MA, USA: MIT Press, 1996. (GECCO '96), p. 357–362. ISBN 0-262-61127-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=1595536-1595586>>.

PERKIS, T. Stack-based genetic programming. In: **Proceedings of the 1994 IEEE World Congress on Computational Intelligence**. Orlando, Florida, USA: IEEE Press, 1994. v. 1, p. 148–153. Disponível em: <<http://citeseer.ist.psu.edu/432690.html>>.

POLI, R. **Discovery of Symbolic, Neuro-Symbolic and Neural Networks with Parallel Distributed Genetic Programming**. [S.l.], aug 1996. Presented at 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97. Disponível em: <<ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSRP-96-14.ps.gz>>.

POLI, R.; LANGDON, W. B. Genetic programming with one-point crossover. In: CHAWDHRY, P. K.; ROY, R.; PANT, R. K. (Ed.). **Soft Computing in Engineering Design and Manufacturing**. Springer-Verlag London, 1997. p. 180–189. ISBN 3-540-76214-0. Disponível em: <<http://cswwww.essex.ac.uk/staff/poli/papers/Poli-WSC2-1997.pdf>>.

_____. On the search properties of different crossover operators in genetic programming. In: KOZA, J. R. et al. (Ed.). **Genetic Programming 1998: Proceedings of the Third Annual Conference**. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 1998. p. 293–301. ISBN 1-55860-548-7. Disponível em: <<http://www.cs.essex.ac.uk/staff/poli/papers/Poli-GP1998.pdf>>.

POLI, R.; LANGDON, W. B.; MCPHEE, N. F. **A field guid to genetic programming**. [S.l.]: Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008. (With contributions by J. R. Koza).

- RUDOLPH, G. Evolutionary search under partially ordered fitness sets. In: **In Proceedings of the International Symposium on Information Science Innovations in Engineering of Natural and Artificial Intelligent Systems (ISI 2001)**. [S.l.]: ICSC Academic Press, 1999. p. 818–822.
- RUMELHART, D. E.; MCCLELLAND, J. L.; GROUP, C. P. R. (Ed.). **Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations**. Cambridge, MA, USA: MIT Press, 1986. ISBN 0-262-68053-X.
- SCHAFFER, J. D. Multiple objective optimization with vector evaluated genetic algorithms. In: **Proceedings of the 1st International Conference on Genetic Algorithms**. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985. p. 93–100. ISBN 0-8058-0426-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=645511.657079>>.
- SMITS, G.; KOTANCHEK, M. Pareto-front exploitation in symbolic regression. In: O'REILLY, U.-M. et al. (Ed.). **Genetic Programming Theory and Practice II**. Ann Arbor: Springer, 2004. cap. 17, p. 283–299. ISBN 0-387-23253-2.
- SRINIVAS, N.; DEB, K. Multiobjective optimization using nondominated sorting in genetic algorithms. **Evol. Comput.**, 1994. MIT Press, Cambridge, MA, USA, v. 2, n. 3, p. 221–248, set. 1994. ISSN 1063-6560. Disponível em: <<http://dx.doi.org/10.1162/evco.1994.2.3.221>>.
- STALLINGS, W. **Arquitetura e organização de computadores: projeto para o desempenho**. 5. ed. São Paulo – SP, Brasil: Prentice Hall, 2002. ISBN 85-87918-53-2.
- TOMASSINI, M. **Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series)**. 1st. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540241930.
- TOMASSINI, M.; CALCOLO, C. S. D. **A Survey of Genetic Algorithms**. 1995.
- TURING, A. M. Computing machinery and intelligence. **Mind**, 1950. Oxford University Press on behalf of the Mind Association, v. 59, n. 236, p. 433–460, 1950. ISSN 00264423.
- VLADISLAVLEVA, E. Y. **Model-based Problem Solving through Symbolic Regression via Pareto Genetic Programming**. Tese (Doutorado) — Universiteit van Tilburg, 2008.
- YAMAMOTO, L.; TSCHUDIN, C. F. Experiments on the automatic evolution of protocols using genetic programming. In: STAVRAKAKIS, I.; SMIRNOV, M. (Ed.). **Autonomic Communication, Second International IFIP Workshop, WAC 2005, Revised Selected Papers**. Athens, Greece: Springer, 2005. (Lecture Notes in Computer Science, v. 3854), p. 13–28. ISBN 3-540-32992-7. Disponível em: <<http://cn.cs.unibas.ch/people/ly/doc/wac2005-lyct.pdf>>.
- ZITZLER, E.; DEB, K.; THIELE, L. Comparison of multiobjective evolutionary algorithms: Empirical results. **Evol. Comput.**, 2000. MIT Press, Cambridge, MA, USA, v. 8, n. 2, p. 173–195, jun. 2000. ISSN 1063-6560. Disponível em: <<http://dx.doi.org/10.1162/106365600568202>>.

ZITZLER, E.; LAUMANN, M.; THIELE, L. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In: GIANNAKOGLU, K. C. et al. (Ed.). **Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems**. Athens, Greece: International Center for Numerical Methods in Engineering, 2001. p. 95–100.

ZITZLER, E.; THIELE, L. Multiobjective evolutionary algorithms: A comparative case study and strength pareto approach. 1999. v. 3, n. 4, 1999.

Apêndices

Exemplificação do *framework fork/join*

Este exemplo visa auxiliar no entendimento da utilização do *framework fork/join* e consiste na aplicação de um feito em uma imagem representada por um vetor de inteiros. Neste vetor, cada posição abriga o valor correspondente à cor do pixel daquela posição. Na imagem resultante, cada pixel é substituído pela média dos valores de imagem dos *pixels* vizinhos. Uma possível solução seria:

```
public class ForkBlur extends RecursiveAction {

    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;
    private int mBlurWidth = 15; // Processing window size,
        should be odd.

    public ForkBlur(int[] src, int start, int length, int[] dst)
    {
        mSource = src;
        mStart = start;
        mLength = length;
        mDestination = dst;
    }

    // Average pixels from source, write results into destination
    .
    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index
            ++){
            // Calculate average.
            float rt = 0, gt = 0, bt = 0;
            for (int mi = -sidePixels; mi <= sidePixels; mi++) {
                int minindex = Math.min(Math.max(mi + index, 0),
                    mSource.length - 1);
                int pixel = mSource[minindex];
```

```

        rt += (float) ((pixel & 0x00ff0000) >> 16) /
            mBlurWidth;
        gt += (float) ((pixel & 0x0000ff00) >> 8) /
            mBlurWidth;
        bt += (float) ((pixel & 0x000000ff) >> 0) /
            mBlurWidth;
    }

    // Re-assemble destination pixel.
    int dpixel = (0xff000000)
        | (((int) rt) << 16)
        | (((int) gt) << 8)
        | (((int) bt) << 0);
    mDestination[index] = dpixel;
}

...

```

A seguir, implementa-se o método abstrato `compute()`, escolhe entre executar a tarefa diretamente ou quebrar a tarefa em duas partes. Um valor mínimo é determinado como limiar.

```

protected static int sThreshold = 10000;

@Override
protected void compute() {
    if (mLength < sThreshold) {
        computeDirectly();
        return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
        new ForkBlur(mSource, mStart + split, mLength - split,
            mDestination));
}

...

```

A seguir, cria-se uma tarefa que representa todo o trabalho a ser feito:

```

protected static int sThreshold = 10000;
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

```

E cria-se o `ForkJoinPool` para executar a tarefa:

```

ForkJoinPool pool = new ForkJoinPool();
pool.invoke(fb);

```

Maiores detalhes podem ser obtidos na documentação oficial da linguagem, no endereço docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

Detalhes na implementação do PPGP

B.1 Diagrama de classe da árvore

A programação genética trabalha evoluindo árvores. Para que a ferramenta seja suficientemente flexível, foi adotada uma organização orientada a objetos que permite à árvore trabalhar todos os tipos de nós, independente se eles encapsulam funções, constantes ou variáveis. Todas as ações e atributos necessários foram colocados na classe `ArrayArvore`.

A Figura 42 mostra o diagrama simplificado da classe `ArrayArvore`, que encapsula um *array* de árvores – cada árvore corresponde a uma saída do problema. Cada árvore encapsula um conjunto de nós definidos pela interface `No`, que é implementada pelas classes `NoFuncao`, `NoConstante` e `NoVariavel`. A classe `NoFuncao` trabalha com elementos da interface `IFuncao`, que pode ser implementada por qualquer tipo de função que se deseje inserir como gene na evolução do sistema de programação genética. É por essa razão que o usuário só necessita conhecer a interface `IFuncao`, como foi apresentado na Seção 5.1.2.2.

B.2 O uso do *framework fork/join*

O *framework fork/join*, apresentado na Seção 4.5.4, foi utilizado para gerar o paralelismo necessário à aplicação. Com esse framework, a ferramenta consegue aumentar o desempenho do sistema se adaptando ao ambiente onde é executado, tudo de maneira transparente ao usuário.

Primeiro, a quantidade de núcleos de processamento presentes no computador em que o programa é executado é automaticamente obtido:

```
int nucleos = Runtime.getRuntime().availableProcessors();
```

Esta informação é utilizada para criar um *pool* de processos, sendo que cada processo é responsável por uma ilha:

```
ForkJoinPool pool = new ForkJoinPool(nucleos);
```

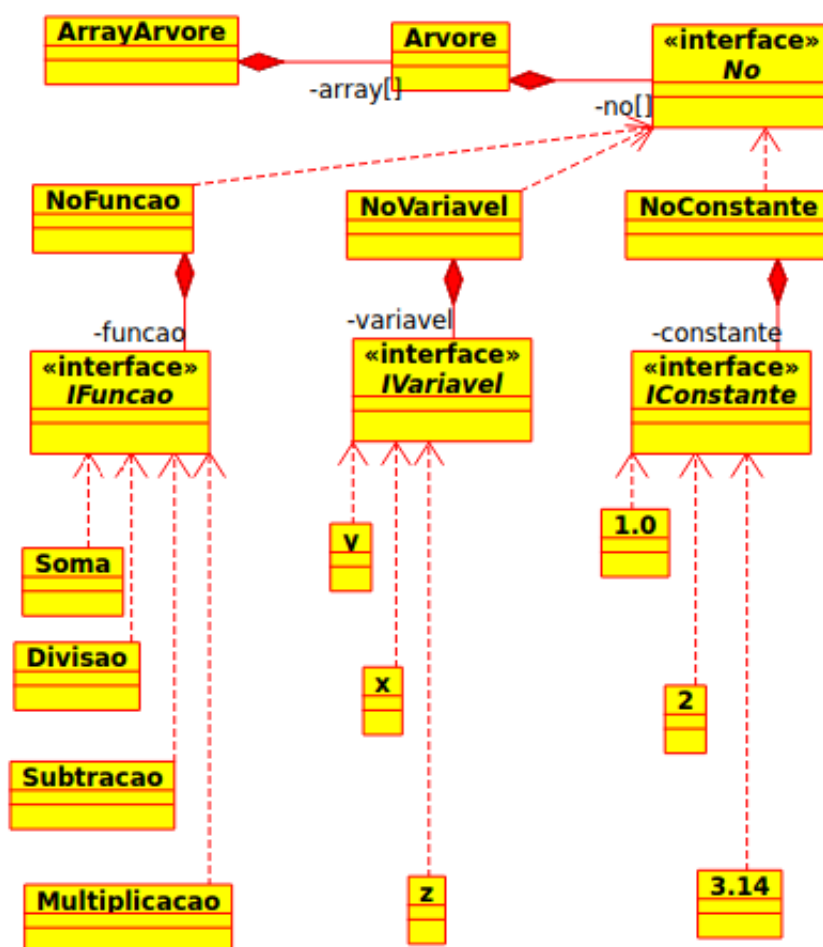


Figura 42 – Organização das classes e interfaces que representam uma árvore.

A população é armazenada em vetores de objetos `ArrayArvore`, que são atribuídos às ilhas de processamento. Dada a informação do tamanho total da população e considerando quantos núcleos estão disponíveis, calcula-se a quantidade de indivíduos farão parte da subpopulação – `tamSubPop`.

A classe `ForkJoinPG` é chamada passando como argumento os valores 0 e o tamanho da população – 1.

```

private class ForkJoinPG extends RecursiveAction {
    private int inicio, fim;
    private int tamSubPop;

    public ForkJoinPG(int inicio, int fim) {
        this.inicio = inicio;
        this.fim = fim;
    }

    @Override
    protected void compute() {
        tamSubPop = fim - inicio + 1;
    }
}

```

```
        if (tamSubPop <= subPop) {
            executaPG(inicio, fim);
            return;
        }

        int meio = (fim + inicio) / 2;
        ForkJoinTask t1 = new ForkJoinPG(inicio, meio);
        ForkJoinTask t2 = new ForkJoinPG(meio + 1, fim);
        invokeAll(t1, t2);
    }
}
```