ERNESTO LUIS MALTA RODRIGUES

EVOLUÇÃO DE FUNÇÕES EM PROGRAMAÇÃO GENÉTICA ORIENTADA A GRAMÁTICAS

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre em Informática no Programa de Pós-Graduação em Informática do Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientadora: Prof.ª Aurora T. R. Pozo

CURITIBA

2002

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno *Ernesto Luis Malta Rodrigues*, avaliamos o trabalho intitulado, ***"Evolução de Funções em Programação Genética Orientada a Gramática"***, cuja defesa foi realizada no dia 19 de abril de 2002, às quatorze horas, no anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 19 de Abril de 2002.

Profª. Drª. Aurora Trinidad Ramirez Pozo
DINF/UFPR - Orientadora

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR

Prof. Dr.
André Carlos Ponce de Leon Ferreira de Carvalho
USP/SC

Prof. Dr. Martin A. Musicante
DINF/UFPR

"... if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the strugle for life; and from the strong principle of inheritance they will tend to produce offspring similary characterized. This principle of preservation, I have called, for the sake of brevity, Natural Selection."

Charles Darwin, 1859

Agradecimentos

Agradeço especialmente a Sandra, minha esposa, e aos meus dois filhos, Marianne e Ernesto Filho, por todo apoio e compreensão.

Agradeço a minha orientadora Profª Aurora, pela orientação e auxílio nos momentos de incerteza.

Agradeço, igualmente, ao Prof Martin e a Profª Silvia que acompanharam este trabalho deste o início e deram contribuições significativas para o seu sucesso.

Não poderia deixar de agradecer, também, aos colegas do grupo de estudos em Computação Evolucionária: Eduardo, Celso, Cláudio, Andréia e Maria Cláudia, pelo companheirismo e dedicação nas fases iniciais desta pesquisa.

Sumário

Índice de Figuras

Índice de Tabelas

Resumo

Um dos grandes desafios da Inteligência Artificial tem sido permitir que os computadores possam resolver problemas sem serem explicitamente programados para tal. Tal desafio implica em um sistema que recebe uma descrição em alto nível de um determinado problema e produz como resposta um programa capaz de resolvê-lo de forma satisfatória.

Neste sentido, a Programação Genética é um método de indução de programas usando, por analogia, os princípios da evolução natural. Durante o processo de evolução, mantém-se uma população de programas que são continuamente alterados. A qualidade de cada programa é avaliada através de uma função de aptidão (*fitness*) que representa o quanto o programa aprendeu a resolver determinado problema. Através da recombinação ou alteração de determinados programas, evolui-se até que a solução seja encontrada ou algum critério de término seja satisfeito.

Tradicionalmente, a formação dos programas é feita através de árvores de sintaxe abstrata, notadamente expressões em LISP. Porém, dependendo do domínio do problema ou da linguagem alvo desejada, algumas restrições devem ser satisfeitas, como por exemplo: formato da solução, tipificação correta dos dados, dentre outras.

Neste trabalho explora-se o uso de gramáticas em Programação Genética como forma de impor restrições sintáticas na formação de programas, orientando a geração de programas e aplicação correta dos operadores genéticos, a fim de produzir programas válidos no domínio desejado.

O uso de evolução de funções também é abordado através da apresentação de diversos métodos atualmente usados (MA, ARL, ADM e ADF). Dentre eles, ADF tem apresentado resultados promissores em problemas cuja complexidade varia em função da escalabilidade, tais como paridade par. Desta forma, ADF foi o método escolhido neste trabalho para verificar se, com o uso de gramáticas, os resultados são semelhantes.

Diferentes aspectos do uso de gramáticas em Programação Genética são discutidos ao longo deste trabalho. Entre eles destacam-se mudança na representação dos programas, alterações na forma de criação da população inicial e restrições na atuação dos operadores genéticos.

A comprovação da eficiência desta abordagem é feita através do uso de uma ferramenta, Chameleon, desenvolvida em C++ e apresentada em detalhes neste trabalho. Chameleon representa uma ferramenta útil de Programação Genética, fácil de usar e adaptável praticamente a qualquer problema que possa ser descrito por uma gramática. Isto é feito sem necessidade de mudanças em seu código fonte, portanto não exigindo do usuário conhecimentos sobre a linguagem em que a ferramenta foi desenvolvida.

Os parâmetros são definidos em um único arquivo texto de fácil compreensão, apesar de Chameleon permitir que sejam modificados durante a execução sem o comprometimento de sua funcionalidade. Além disso, permite a gravação e recuperação de populações durante o processo evolutivo, permitindo o uso de *checkpoints*.

A eficiência da abordagem é comparada com os resultados apresentados por John Koza no problema de paridade par (*even parity*). Pelos resultados obtidos até agora, o uso de gramáticas apresenta *performance* equivalente e diversas vantagens como: independência de domínio, uso de qualquer linguagem alvo para solução e facilidade em se impor o formato da solução.

Abstract

One of the great challenges of Artificial Intelligence is to get a computer to solve a problem without explicitly programming it. This challenge envisions an automatic system whose input is a high level statement of a problem's requirements and whose output is a satisfactory solution to the given problem.

The Genetic Programming method is an approach to inductively forming programs. The use of natural selection based on a fitness function for reproduction of the program population has allowed many problems to be solved.

Traditionally, the programs are represented by an abstract syntax tree, usually LISP S-expressions. However, depending on the problem or the desired language, some restrictions must be satisfied, to ensure legal programs are always created.

In this work, we explore the use of grammars in Genetic Programming to define the structure of the initial population and to direct crossover and mutation operators to produce only valid programs in the desired domain.

The evolution of functions are also presented. Differents methods proposed by various researches (MA, ARL, ADM and ADF) are described. Among them, ADF is advantageous for the scaled-up versions of the even-parity problem. In this way, we study how well the combination of grammars and ADF solves a even-parity problem.

We detail how to modify the Genetic Programming components to accommodate a grammar, focusing the program representation, the initialization procedure and genetic operators constraints.

A Grammar-Guided Genetic Programming system called Chameleon are presented. The system was written in C++ and its implementation details are discussed. Chameleon is a valuable tool to work with the Genetic Programming and grammars.

The parameters are defined in a simple text file, although Chameleon allows parameter changes during the run, without lost of functionality. Furthermore, a population can be saved or loaded during the evolution process, allowing the use of checkpoints.

The efficiency of the grammar approach is compared with the results obtained by John Koza in the even parity problem. We observed that the use of grammars does not improve the convergency, but it has many advantages such as: domain independence, easiness to impose the format of the solution, among others.

1. INTRODUÇÃO

A Computação Evolucionária compreende um conjunto de técnicas de busca e otimização inspiradas na evolução natural das espécies. Desta forma, cria-se uma população de indivíduos que vão reproduzir e competir pela sobrevivência. Os melhores sobrevivem e transferem suas características a novas gerações. As técnicas atualmente incluem [Banzhaf 1998]: Programação Evolucionária, Estratégias Evolucionárias, Algoritmos Genéticos e Programação Genética. Estes métodos estão sendo utilizados, cada vez mais, pela comunidade de inteligência artificial para obter modelos de inteligência computacional [Barreto 1997].

O paradigma da Programação Genética foi desenvolvido por John Koza [Koza 1989; Koza 1992] com base nos trabalhos de John Holland em Algoritmos Genéticos [Holland 1975]. Atualmente representa uma área muito promissora de pesquisa em Inteligência Artificial devido a sua simplicidade e robustez. Seu uso tem sido estendido a problemas de diversas áreas do conhecimento, como por exemplo: biotecnologia, engenharia elétrica, análises financeiras, processamento de imagens, reconhecimento de padrões, mineração de dados, linguagem natural, dentre muitas outras [Willis 1997]

A Programação Genética é a evolução de um conjunto de programas com o objetivo de aprendizagem por indução [Banzhaf 1998]. A idéia é ensinar computadores a se programar, isto é, a partir de especificações de comportamento, o computador deve ser capaz de induzir um programa que as satisfaça [Koza 1992]. A cada programa é associado um valor de mérito (*fitness*) representando o quanto ele é capaz de resolver o problema.

Basicamente, a Programação Genética mantém uma população de programas de computador, usa métodos de seleção baseados na capacidade de adaptação (*fitness*) de cada programa (escolha dos "melhores"), aplica operadores genéticos para modificá-los e convergir para uma solução. O objetivo é encontrar uma solução no espaço de todos os programas possíveis (candidatos) usando apenas um valor de *fitness* como auxílio no processo de busca [Gathercole 1998].

O mecanismo de busca da Programação Genética pode ser descrito como um ciclo "criar-testar-modificar" (Figura 1), muito similar a forma com que os humanos desenvolvem seus programas. Inicialmente, programas são criados baseados no conhecimento sobre o domínio do problema. Em seguida, são testados para verificar sua funcionalidade. Se os resultados não forem satisfatórios, modificações são feitas para melhorá-los. Este ciclo é repetido até que uma solução satisfatória seja encontrada ou um determinado critério seja satisfeito [Yu 1999].

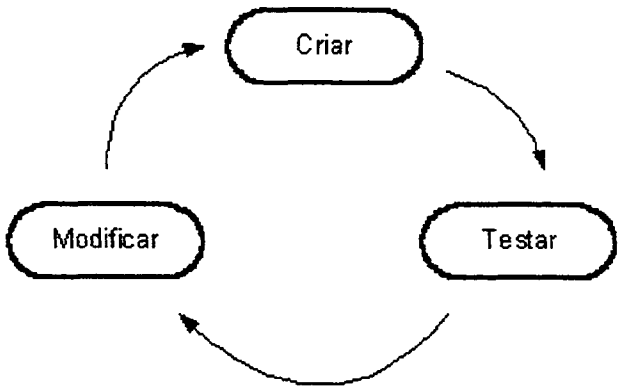

FIGURA 1: CICLO "CRIAR-TESTAR-MODIFICAR"

A especificação de comportamento é feita normalmente através de um conjunto de valores de entrada-saída, denominados *fitness cases*, representando o conjunto de aprendizagem ou treinamento (*training set*). Com base neste conjunto, a Programação Genética procura obter um programa que: [O'Reilly 1995]

- Produza, de forma não trivial, as saídas corretas para cada entrada fornecida. Isto implica que o programa não deve mapear as entradas e saídas através de alguma forma de tabela de conversão. Portanto, o programa deverá aprender necessariamente alguma forma de algoritmo;

- Calcule as saídas de tal forma que, se as entradas forem representativamente escolhidas, o programa será capaz de produzir saídas corretas para entradas não cobertas inicialmente.

Por manipular programas diretamente, a Programação Genética lida com uma estrutura relativamente complexa e variável. Tradicionalmente, esta estrutura é uma árvore de sintaxe abstrata composta por funções em seus nós internos e por terminais em seus nós-folha. A especificação do domínio do problema é feita simplesmente pela definição dos conjuntos de funções e terminais [Koza 1992].

Contudo, muitos problemas impõem restrições de construção sintática dos programas. Um exemplo é o caso das Séries de Fourier que apresentam um formato pré-definido para a solução[1]. Para solucionar este tipo de problema, John Koza desenvolveu o conceito de Estruturas Sintáticas Restritas[2] (*Constraint Syntatic Structures*) [Koza 1992]. Em seu trabalho, John Koza aplicou esta abordagem a diversos problemas tais como: regressão múltipla, somador lógico de dois bits (*two-bit adder*), sistema de equações lineares, Série de Fourier, redes neurais, dentre outros. Para cada tipo de problema, as modificações necessárias no algoritmo foram apresentadas. Porém a abordagem é muito limitada pois exige mudanças no algoritmo para cada restrição a satisfazer [Whigham 1995].

Para lidar com domínios com restrição de tipo de dados, David Montana [Montana 1994; Montana 1995] desenvolveu a Programação Genética Fortemente Tipada (*Strongly Typed Genetic Programming*, STGP) onde cada função é caracterizada pelos tipos de dados que pode manipular e cada terminal tem seu tipo de dado definido. Desta forma, a geração e evolução dos programas respeitam as restrições de tipo facilmente. Infelizmente, esta abordagem não permite também restringir o formato dos programas [Whigham 1995].

O uso de gramáticas em Programação Genética foi inicialmente proposto por Frederic Gruau [Gruau 1995; Gruau 1996] como uma forma de impor restrições sintáticas à geração e preservação dos programas, restringindo o espaço de busca. A especificação das regras da gramática é informada como parte dos parâmetros a serem usados pelo algoritmo. Através desta abordagem, não há a necessidade de modificar o algoritmo cada vez que as restrições sintáticas são modificadas.

---

[1] O Problema das Séries de Fourier será abordado na Seção 2.10 (pág. 24)
[2] Capítulo 19 - *Evolution of Constrained Syntax Structures* [Koza 1992]

Peter Whigham [Whigham 1995; Whigham 1996] demonstrou que as gramáticas também podem funcionar como uma forma de indução (*bias*) no processo evolutivo, facilitando a descoberta do formato da solução. A técnica se baseia no fato de privilegiar as regras de formação que geram melhores programas e descobrir novas regras que facilitem a convergência.

Por outro lado, é comum em programação o uso de módulos e funções. Elas estimulam a abordagem *top-down* que facilita a forma de se resolver problemas mais complexos, gerando soluções de forma hierárquica [Koza 1994]. A técnica se baseia na subdivisão de um problema em problemas menores, mais fáceis de implementar [Terada 1991]. Em Programação Genética, várias abordagens de criação de funções e módulos foram propostas: MA (*Module Acquisition*) [Angeline 1993], ADF (*Automatically Defined Functions*) [Koza 1994], ADM (*Automatically Defined Macros*) [Spector 1996] e ARL (*Adaptative Representation through Learning*) [Rosca 1996; Rosca 1997].

O método *Module Acquisition* (MA) se baseia na idéia de se criar automaticamente uma "biblioteca" de funções usando partes de programas. Estas funções são então livremente usadas por qualquer programa, porém não sofrem ação dos operadores genéticos (são funções "estáveis") [Kinnear Jr 1994]

A abordagem das *Automatically Defined Functions* (ADFs) se baseia na construção dos programas juntamente com funções. Cada programa tem o seu conjunto de uma ou mais funções. Cada função, diferentemente de MA, é usada unicamente pelo programa que a definiu e pode sofrer a ação dos operadores genéticos (as funções podem evoluir) [O'Reilly 1996]

As *Automatically Defined Macros* (ADM) estabelecem uma forma de macrosubstituição. Quando durante a avaliação de um programa houver uma chamada a uma ADM, ela é expandida dentro do contexto do programa. O código gerado propriamente dito depende das condições no momento da sua execução. Esta abordagem tem se demonstrado interessante quando envolve funções com efeitos colaterais[3], tais como movimentação de robôs [Spector 1996].

[3] Função com efeito colateral é aquela que interfere em elementos externos à função propriamente dita.

De forma semelhante ao MA, o método *Adaptative Representation through Learning* (ARL) extrai partes de programas para criar funções. Todavia, ao invés de criar uma "biblioteca", estas funções são adicionadas ao conjunto de funções, de forma a serem usadas nas próximas gerações. Se elas forem úteis, serão preservadas. Caso contrário, são eliminadas. Desta forma, o conjunto de funções expande e contrai durante o processo evolutivo. Recentemente, este método foi criticado por sua fraca capacidade em produzir soluções de forma hierárquica [Dessi 1999].

Destes métodos, ADF é de especial interesse neste trabalho devido ao fato de ter sido aplicado a um conjunto relativamente amplo de problemas com sucesso [Koza 1994].

1.1 Objetivos do Trabalho

Este trabalho tem dois objetivos. Primeiramente, apresentar uma abordagem para Programação Genética com uso de gramáticas. Isto é feito através de Chameleon, uma ferramenta capaz de evoluir programas baseados em uma gramática. Segundo, estender esta abordagem para suportar ADF e demonstrar que, mesmo com o uso de gramáticas, ADF facilita a descoberta da solução para problemas mais complexos tais como o paridade par.

1.2 Contribuições

A implementação de gramáticas em Programação Genética exige mudanças na forma com que os programas são gerados e manipulados. Os programas devem sempre respeitar as regras da gramática, portanto a exploração do espaço de busca deve ser controlada. Uma ferramenta de Programação Genética com uso de gramáticas deve apresentar mecanismos genéricos de criação e manipulação de programas com um forte comprometimento com a sua consistência sintática.

Desta forma, a principal contribuição deste trabalho é a apresentação de Chameleon, uma ferramenta de Programação Genética baseada em gramáticas com suporte a evolução de funções.

Chameleon representa uma ferramenta útil de Programação Genética por permitir o uso de gramáticas e ADF. Ferramentas de domínio público em C/C++, tais como GPC++ [Fraser 1994], LILGP [Zongker 1996], GP-QUICK [Langdon 1996], SYSGP [Brameier 1998] e, mais recentemente, GAPS [Kramer 2000], não contemplam o uso de gramáticas e geram expressões em LISP somente. Apenas a GPK de Helmut Horner [Horner 1996] apresenta suporte a gramáticas, porém não permite uso de ADF.

1.3 Organização do Trabalho

A dissertação está dividida, contando com esta introdução, em sete capítulos. Nesta introdução, apresentou-se uma visão geral do trabalho, sua motivação e seus objetivos. No Capítulo 2, a Programação Genética tradicional[4] é apresentada em detalhes. Inicialmente aborda-se sua origem e uma visão geral de seu método de busca, juntamente com a representação dos programas. Em seguida, os diversos métodos de criação, avaliação e seleção dos programas são abordados e doravante mostrar a atuação dos operadores genéticos. A última parte do capítulo traz uma análise das limitações da abordagem tradicional.

O uso de gramáticas para guiar a criação de programas e auxiliar a atuação dos operadores genéticos é apresentado no Capítulo 3. As modificações necessárias na representação dos programas e na atuação dos operadores genéticos são discutidas no decorrer do capítulo.

Uma visão geral de evolução de funções com suas diferentes abordagens são apresentadas no Capítulo 4. Uma ênfase maior é dada no uso de ADF e sua aplicação com gramáticas. Ao final do capítulo, apresenta-se a abordagem de representação de ADF através de gramáticas adotada neste trabalho.

[4] Entende-se por Programação Genética tradicional a que foi introduzida por John Koza [Koza 1992]

O Capítulo 5 é dedicado a uma descrição da ferramenta Chameleon, através de seus diagramas funcional e de classes. Seu funcionamento é descrito em detalhes e diversos exemplos de execução são apresentados.

Os experimentos e seus resultados são abordados e analisados no Capítulo 6 para, finalmente, no Capítulo 7, serem apresentadas as conclusões e trabalhos futuros, enfatizando-se as principais contribuições do trabalho.

2. PROGRAMAÇÃO GENÉTICA

Neste capítulo é fornecida uma explicação sobre a origem, funcionamento e principais componentes da Programação Genética e algumas de suas extensões. Vários aspectos importantes são detalhados, tais como a geração da população inicial e avaliação dos programas.

2.1 Origem

A Programação Genética é uma abordagem para a geração automática de programas de computador desenvolvida por John Koza [Koza 1989; Koza 1992]. A técnica se baseia na combinação de idéias da teoria da evolução (seleção natural), genética (reprodução, cruzamento e mutação), inteligência artificial (busca heurística) e teoria de compiladores (representação de programas como árvores sintáticas). Basicamente, a Programação Genética é um algoritmo que busca, dentre um espaço relativamente grande porém restrito de programas de computador, uma solução ou, pelo menos, uma boa aproximação para resolver determinado problema [Bruce 1995]

A teoria da seleção natural das espécies defende que, na natureza, sobrevivem os seres que têm a melhor capacidade de se adaptarem às mudanças que ocorrem no meio ambiente. Estes indivíduos passarão suas características genéticas para seus descendentes e, ao final de muitas gerações, obtem-se uma população de indivíduos com estas características selecionadas naturalmente.

Em vez de uma população de seres vivos, em Programação Genética tem-se uma população de programas de computador. O objetivo do algoritmo é, através da evolução destes programas, chegar a um programa que melhor resolva um determinado problema. Para isso, começa-se com uma população inicial aleatória e, geração após geração, aplica-se os operadores genéticos para simular o processo evolutivo até que um determinado critério de término seja satisfeito.

2.2 Visão Geral do Algoritmo de Programação Genética

O algoritmo de Programação Genética é simples e pode ser descrito resumidamente como:

- Criar aleatoriamente[5] uma população de programas;
- Executar os seguintes passos até que um Critério de Término seja satisfeito:
    - Avaliar cada programa através de uma função heurística (*fitness*), que expressa quão próximo cada programa está da solução ideal;
    - Selecionar os melhores programas de acordo com o *fitness*;
    - Aplicar a estes programas os operadores genéticos (reprodução, cruzamento e mutação)
- Retornar com o melhor programa encontrado

Cada execução deste laço representa uma nova geração de programas. Tradicionalmente, o *Critério de Término* é estabelecido como sendo encontrar uma solução satisfatória ou atingir um número máximo de gerações [Koza 1992]. Porém, existem abordagens baseadas na análise do processo evolutivo, isto é, o laço permanece enquanto houver melhoria na população [Kramer 2000].

A estrutura básica do algoritmo de Programação Genética é mostrada na Figura 2.

[5] A geração inicial representa uma "busca cega" pela solução.

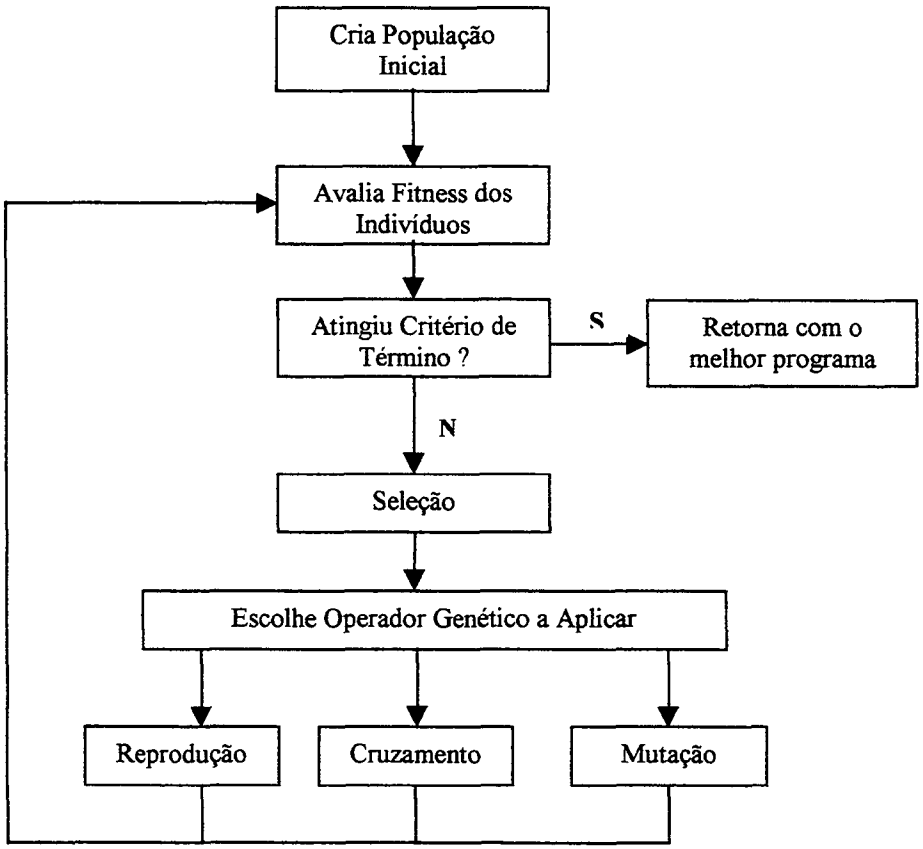

FIGURA 2: ESTRUTURA BÁSICA DO ALGORITMO DE PROGRAMAÇÃO GENÉTICA.

2.3 Representação dos Programas

A representação dos programas em Programação Genética tradicionalmente se baseia em árvore de sintaxe abstrata, isto é, os programas são formados pela livre combinação de funções e terminais adequados ao domínio do problema.

Parte-se de dois conjuntos: F como sendo o conjunto de funções e T como o conjunto de terminais. O conjunto F pode conter operadores aritméticos (+, -, * etc.), funções matemáticas (`seno`, `log` etc.), operadores lógicos (`E`, `OU` etc.) dentre outros. Cada $f \in F$ tem associada uma aridade (número de argumentos) superior a zero. O conjunto T é composto pelas variáveis, constantes e funções de aridade zero (sem argumentos).

Por exemplo, considerando o conjunto dos operadores aritméticos de aridade dois (2) como sendo o conjunto de funções e a variável x e a constante dois (2) como terminais, isto é:

$$F = \{ +, -, *, / \}$$

$$T = \{ x, 2 \}$$

então expressões matemáticas simples tais como `x*x+2` podem ser produzidas. A representação é feita por uma árvore de sintaxe abstrata como mostrado na Figura 3.

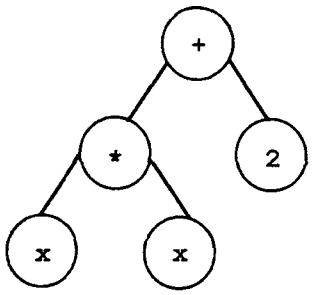

FIGURA 3: ÁRVORE DE SINTAXE ABSTRATA DE `x*x+2`

O espaço de busca é determinado por todas as árvores que possam ser criadas pela livre combinação de elementos dos conjuntos F e T.

2.4 Fechamento e Suficiência

Para garantir a viabilidade das árvores de sintaxe abstrata, John Koza definiu a propriedade de Fechamento (*closure*) [Koza 1992]. Para satisfazê-la, cada função do conjunto F deve aceitar, como seus argumentos, qualquer valor que possa ser retornado por qualquer função ou terminal. Esta imposição garante que qualquer árvore gerada pode ser avaliada corretamente.

Um caso típico de problema de Fechamento é a operação de divisão. Matematicamente, não é possível dividir um valor por zero. Uma abordagem possível é definir uma função alternativa que permita um valor para a divisão por zero. É o caso da função de divisão protegida (*protected division*) % proposta por [Koza 1992]. A função % recebe dois argumentos e retorna o valor 1 (um) caso seja feita uma divisão por zero e, caso contrário, o seu quociente.

Para garantir a convergência para uma solução, John Koza definiu a propriedade de Suficiência (*sufficiency*) onde os conjuntos de funções F e o de terminais T devem ser capazes de representar uma solução para o problema [Koza 1992]. Isto implica que deve existir uma forte evidência de que alguma composição de funções e terminais possa produzir uma solução. Dependendo do problema, esta propriedade pode ser óbvia ou exigir algum conhecimento prévio de como deverá ser a solução.

2.5 População Inicial

Tradicionalmente, a população inicial é composta por árvores geradas aleatoriamente a partir dos conjuntos de funções F e de terminais T. Inicialmente se escolhe aleatoriamente uma função $f \in F$. Para cada um dos argumentos de f, escolhe-se um elemento de $\{ F \cup T \}$. O processo prossegue até que se tenha apenas terminais como nós-folha da árvore. Usualmente se especifica um limite máximo para a profundidade da árvore para se evitar árvores muitos grandes.

Porém, a "qualidade" da população inicial é um fator crítico para o sucesso do processo evolutivo [Daida 1999]. A população inicial deve ser uma amostra significativa do espaço de busca, apresentando uma grande variedade de composição nos programas, para que seja possível, através da recombinação de seus códigos, convergir para uma solução.

Para melhorar a qualidade dos programas gerados na população inicial, há diversos métodos, sendo os mais comuns [Luke 2001]: *ramped-half-and-half* [Koza 1992], *random-branch* [Chellapilla 1997], *uniform* [Bohm 1996] e, mais recentemente, *probabilistic tree-creation* [Luke 2000].

O método *ramped-half-and-half* [Koza 1992] é uma combinação de dois métodos simples: *grow* e *full*.

O método *grow* envolve a criação de árvores cuja profundidade[6] é variável. A escolha dos nós é feita aleatoriamente entre funções e terminais, respeitando-se uma profundidade máxima. O algoritmo é muito simples e está na Figura 4.

```
Dados:
        Profundidade máxima D
        Conjunto de funções F e de terminais T
Faça:
        Árvore = GROW(0)

GROW(profundidade d)
        Retorna: uma árvore de profundidade máxima ≤ D - d

        Se d = D
                Retorne aleatoriamente um terminal de T
        Senão
                Escolha aleatoriamente um elemento e ∈ { F ∪ T }
                Se e ∈ F
                        Para cada argumento a de e
                                Preencha a com GROW(d + 1)

        Retorne f com todos os argumentos preenchidos
```

FIGURA 4: ALGORITMO GROW

Já o método *full* envolve a criação de árvores completas, isto é, todas as árvores terão a mesma profundidade. Isto é facilmente feito através da seleção de funções para os nós cuja profundidade seja inferior a desejada e a seleção de terminais para os nós de profundidade máxima.

Combinar os métodos *full* e *grow* com objetivo de gerar um número igual de árvores para cada profundidade, entre dois e a profundidade máxima, é a base do método *ramped-half-and-half* [Koza 1992]. Por exemplo, supondo que a profundidade máxima seja seis, então serão geradas árvores com profundidades de dois, três, quatro, cinco e seis eqüitativamente. Isto significa que 20% terão profundidade dois, 20% terão

[6] A profundidade de um nó *n* em uma árvore é o comprimento de caminho da raiz até *n*. A profundidade de uma árvore é o nó de maior profundidade [Terada 1991]

profundidade três e assim sucessivamente. Para cada profundidade, 50% são geradas pelo método *full* e 50% pelo método *grow*.

As desvantagens deste método são [Luke 2000]:

- Impõe uma faixa fixa de profundidades (normalmente entre 2 e 6), independentemente do tamanho da árvore. Dependendo do número de argumentos (aridade) de cada função, mesmo com a mesma profundidade, podem ser geradas árvores de tamanhos[7] muito diferentes;

- A escolha da profundidade máxima, antes de se gerar a árvore, não é aleatória e sim de forma proporcional;

- Se o conjunto de funções for maior que o de terminais (como na maioria dos problemas), a tendência é gerar a maior árvore possível ao aplicar *grow*;

O método *random-branch* [Chellapilla 1997] permite que se informe qual o tamanho máximo da árvore (e não a sua profundidade). O algoritmo está na Figura 5.

```
RANDOM-BRANCH(tamanho máximo desejado S)
    Retorna: uma árvore de tamanho ≤ S

    Se um não-terminal com aridade ≤ S não existe
        Retorne aleatoriamente um terminal
    Senão
        Escolha aleatoriamente um não-terminal n com aridade ≤ S
        Seja b_n a aridade de n
        Para cada argumento a de n
            Preencha a com RANDOMBRANCH(⌊a / b_n⌋)
    Retorne n com todos os argumentos preenchidos
```

FIGURA 5: ALGORITMO RANDOM-BRANCH

[7] O tamanho de uma árvore é o número de nós que a compõem [Koza 1992].

Porém, devido ao fato de *random-branch* dividir igualmente S dentre as árvores de um nó-pai não-terminal, existem muitas árvores que não são possíveis de serem produzidas. Isto torna o método muito restritivo apesar de ter complexidade linear [Luke 2001].

O método *uniform* foi desenvolvido por Bohm com o objetivo de garantir que as árvores são geradas uniformemente do conjunto de todas as árvores possíveis [Bohm 1996]. O algoritmo é extremamente complexo, pois necessita calcular em várias tabelas o número de árvores possíveis de serem geradas para cada tamanho desejado. A desvantagem deste método é o seu alto custo computacional. Um exemplo de aplicação deste método é a ferramenta GPK de Helmut Horner [Horner 1996].

Os métodos *probabilistic tree-creation* (PTC) 1 e 2 [Luke 2000], ao contrário dos outros métodos, não procuram gerar estruturas de árvores completamente uniformes. Ao invés disso, permite definir as probabilidades de ocorrência das funções na árvore.

O PTC1 é uma variante do *grow* onde para cada terminal $t \in T$, associa-se uma probabilidade q_t dele ser escolhido quando houver necessidade de um terminal. O mesmo se faz com cada $f \in F$, associando-se uma probabilidade q_f. Antes de gerar qualquer árvore, o algoritmo calcula p, a probabilidade de escolher um não-terminal ao invés de um terminal, de forma a produzir uma árvore de tamanho esperado E_{tree}. A obtenção do valor de p é feita pela fórmula a seguir:

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n} \quad \text{onde } b_n \text{ é a aridade do não-terminal } n \qquad (1)$$

O algoritmo do PTC1 está na Figura 6.

```
Dados:
        Profundidade máxima D
        Conjunto de funções F e de terminais T
        A probabilidade p de escolher uma função
        As probabilidades q_t e q_f para cada t ∈ T e f ∈ F
Faça:
        Árvore = PTC1(0)

PTC1(profundidade d)
        Retorna: uma árvore de profundidade máxima ≤ D - d

        Se d = D
                Retorne aleatoriamente um terminal de T baseado em q_t
        Senão
                Com probabilidade p, Se uma função deve ser escolhida
                        Escolha aleatoriamente f ∈ F baseado em q_f
                        Para cada argumento a de e
                                Preencha a com PTC1(d + 1)
                        Retorne f com todos os argumentos preenchidos
                Senão
                        Retorne aleatoriamente um terminal de T baseado em q_t
```

FIGURA 6: ALGORITMO PTC1

PTC1 garante que as árvores serão geradas dentro de um tamanho esperado. Uma variante deste método (PTC2) usa um tamanho máximo S e uma distribuição de probabilidades $w_1, w_2, ...w_s$ para cada árvore de tamanho 1 a S. Além do controle sobre o tamanho esperado da árvore, tem-se um controle sobre a distribuição destes tamanhos.

2.6 Função de Aptidão

Na natureza os seres vivos são selecionados naturalmente com base no seu grau de adaptabilidade ao meio ambiente. Em Programação Genética, isto é expresso pela função de aptidão ou *fitness*. Os programas que melhor resolverem o problema receberão melhores valores de *fitness* e, consequentemente, terão maior chance de serem selecionados para reproduzir.

A avaliação de *fitness* depende do domínio do problema e pode ser medida de diversas formas, tanto direta quanto indiretamente. Para fins deste trabalho, apenas os domínios que permitam uma avaliação direta de *fitness* são considerados..

Usualmente, para se proceder à avaliação de *fitness*, é fornecido um conjunto de casos de treinamento, denominados *fitness cases*, contendo valores de entrada e saída a serem aprendidos. A cada programa é fornecido os valores de entrada e confronta-se a sua resposta ao valor esperado de saída. Quanto mais próxima a resposta do programa estiver do valor de saída, melhor é o programa.

Desta forma, a avaliação de *fitness* estabelece uma forma de se diferenciar os melhores dos piores, servindo como a força mestre do processo evolutivo, sendo a medida (usada durante a evolução) do quanto o programa aprendeu a predizer as saídas das entradas dentro de um domínio de aprendizagem [Banzhaf 1998].

A escolha da função de *fitness*, assim como a escolha do método de avaliação utilizado por esta função, depende do problema. Boas escolhas são essenciais para se obterem bons resultados, já que a função de *fitness* é a força-guia que direciona o algoritmo de Programação Genética na busca pela solução [Gritz 1993].

Os métodos comumente usados para avaliação de *fitness* são [Koza 1992]:

1) **Aptidão nata** (*raw fitness*): representa a medida dentro do próprio domínio do problema. É a avaliação pura e simples do programa frente aos *fitness cases*. O método mais comum de aptidão nata é a avaliação do erro cometido, isto é, a soma de todas as diferenças absolutas entre o resultado obtido pelo programa e o seu valor correto.

2) **Aptidão padronizada** (*standardized fitness)*: Devido ao fato da aptidão nata depender do domínio do problema, uma valor bom pode ser um valor pequeno (quando se avalia o erro) ou um valor grande (quando se avalia a taxa de eficiência). A avaliação da aptidão padronizada é feita através de uma função de adaptação do valor da aptidão nata de forma que quanto melhor o programa, menor deve ser a

aptidão padronizada . Desta forma, o melhor programa apresentará o valor zero (0) como aptidão padronizada, independentemente do domínio do problema.

3) **Aptidão ajustada** (*adjusted fitness*): é obtida através da aptidão padronizada. Se *s(i, t)* representa a aptidão padronizada do indivíduo *i* na geração *t*, então a aptidão ajustada *a(i, t)* é calculada da seguinte forma:

$$a(i,t) = \frac{1}{1+s(i,t)} \qquad (2)$$

Percebe-se que a aptidão ajustada varia entre zero (0) e um (1), sendo que os maiores valores representam os melhores indivíduos. A aptidão ajustada tem o benefício de exagerar a importância de pequenas diferenças no valor da aptidão padronizada quando esta se aproxima de zero [Koza 1992].

4) **Aptidão normalizada** *(normalized fitness)*: se *a(i, t)* é a aptidão ajustada do indivíduo *i* na geração *t*, então sua aptidão normalizada *n(i, t)* será obtida da seguinte forma:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{m} a(k,t)} \qquad (3)$$

É fácil perceber que a soma de todas as aptidões normalizadas dentro de uma população vale um (1).

Para uma melhor comprensão de como pode ser feita a avaliação de *fitness*, suponha os seguintes valores de *fitness cases* mostrados na Tabela 1.

	ENTRADA	SAÍDA
FITNESS CASE 1	0	1
FITNESS CASE 2	2	5
FITNESS CASE 3	4	17
FITNESS CASE 4	6	37
FITNESS CASE 5	8	65

TABELA 1: CONJUNTO DE FITNESS CASES

Com base neste conjunto, deseja-se descobrir um programa que seja capaz de produzir as saídas para cada entrada informada. É fácil perceber que a função $f(x) = x^2+1$ é uma solução válida neste caso.

A aptidão nata (*raw fitness*) para este tipo de problema pode ser a soma das diferenças absolutas da resposta do programa pela saída correta (*Minkowski distance*). Para cada programa p pertencente a população P, associa-se um valor f_p que representa o seu *fitness* obtido na avaliação dos n *fitness cases* informados. O valor de f_p é obtido pela fórmula:

$$f_p = \sum |p_i - s_i| \qquad (4)$$

Onde p_i representa a resposta do programa ao i-ésimo *fitness case* e s_i, a saída correta. Quanto mais perto o valor de p_i estiver de s_i, menor será o valor de f_p e melhor será o programa. Neste caso, esta avaliação de *fitness* também é considerada uma aptidão padronizada (*standardized fitness*).

Em algumas aplicações, é interessante reforçar a diferença entre os diversos valores de *fitness* de uma população. Uma variação muito comum é somar o quadrado das diferenças (*squared error*), como a fórmula a seguir:

$$f_p = \sum_{i=1}^{n} (p_i - s_i)^2 \qquad (5)$$

Qual o real impacto do uso destas funções de *fitness*? Para melhor esclarecer, considere que o programa $x^2 + x$ está sendo avaliado. A Tabela 2 mostra os resultados obtidos em cada uma das funções para os valores da Tabela 1.

	ENTRADA	SAÍDA	PROGRAMA	ERRO ABSOLUTO	ERRO QUADRÁTICO
FITNESS CASE 1	0	1	0	1	1
FITNESS CASE 2	2	5	6	1	1
FITNESS CASE 3	4	17	20	3	9
FITNESS CASE 4	6	37	42	5	25
FITNESS CASE 5	8	65	72	7	49
Valor de *fitness*				17	85

TABELA 2: DOIS MÉTODOS DE CÁLCULO DE FITNESS

As duas formas de avaliação de *fitness* apresentadas são adequadas quando o comportamento do programa pode ser descrito através do conjunto de *fitness cases*, isto é, uma tabela de valores de entrada e saída. Uma categoria típica destes problemas é a Regressão Simbólica.

2.7 Métodos de Seleção

O método de seleção tem por objetivo escolher quais programas deverão sofrer a ação dos operadores genéticos e compor uma nova geração. Dado que a "qualidade" de um programa é dada pelo seu valor de *fitness*, a seleção deve preferenciar, de alguma forma, os programas que apresentem os melhores valores de *fitness*.

Os métodos atualmente usados são [Blickle 1995]: Seleção Proporcional, Seleção por Torneio, Seleção por Truncamento, Seleção por Nivelamento Linear e Seleção por Nivelamento Exponencial.

1) **Seleção Proporcional** (*fitness-proportionate selection*): Apresentada por John Holland [Holland 1975] para Algoritmos Genéticos, foi o método escolhido por John Koza no seu primeiro livro [Koza 1992]. Usa a aptidão normalizada disposta em uma "roleta", sendo que cada indivíduo da população ocupa uma "fatia" proporcional a sua aptidão normalizada. Em seguida é produzido um número aleatório entre zero (0) e um (1). Este número representará a posição ocupada pela "agulha" da roleta. Apesar de seu grande sucesso devido a sua simplicidade, este método é muito afetado pela escalabilidade da aptidão normalizada [Blickle 1995].

2) **Seleção por Torneio** (*tournament selection*): Apresentada por David Goldberg [Goldberg 1991] para Algoritmos Genéticos, foi utilizada em vários problemas por John Koza no seu segundo livro [Koza 1994]. A seleção por torneio é feita da seguinte forma: t indivíduos são escolhidos aleatoriamente da população e o melhor deles é o escolhido. Este processo é repetido até que se tenha uma nova população. O valor de t é conhecido como o tamanho do torneio.

3) **Seleção por Truncamento** (*truncation selection*): Com base em um valor de limiar (*threshold*) T entre zero (0) e um (1), a seleção é feita aleatoriamente entre os T melhores indivíduos [Muhlenbein 1993]. Por exemplo, se $T = 0.4$, então a seleção é feita entre os 40 % melhores indivíduos e os outros 60 % são descartados.

4) **Seleção por Nivelamento Linear** (*linear ranking selection*): Sugerido por Baker [Baker 1989] para eliminar as sérias desvantagens do uso de seleção proporcional. Para tal, os indivíduos são ordenados de acordo com os valores de *fitness* e o nível N é associado ao melhor indivíduo e o nível 1, ao pior. Em seguida, a cada indivíduo i é associada uma probabilidade p_i de ser selecionado.

$$p_i = \frac{1}{N}\left(n^- + \left(n^+ - n^-\right)\frac{i-1}{N-1}\right) \quad \text{onde } i \in \{1,2,...N\}, \quad n^- \geq 0 \text{ e } n^+ + n^- = 2 \qquad (6)$$

O valor de $\frac{n^+}{N}$ representa a probabilidade do melhor indivíduo ser escolhido e $\frac{n^-}{N}$, a do pior ser escolhido. É interessante perceber que cada indivíduo pertence a um único nível, isto é, mesmo que dois indivíduos tenham o mesmo *fitness*, eles apresentam probabilidades diferentes de serem escolhidos.

5) **Seleção por Nivelamento Exponencial (*exponential ranking selection*):** A seleção por nivelamento exponencial se diferencia do Nivelamento Linear apenas no fato das probabilidades p_i serem exponencialmente ponderadas [Baker 1989]. Um parâmetro c entre zero (0) e um (1) é usado como base. Quanto mais próximo de um, menor é a "exponencialidade" da seleção. Tal como no Nivelamento Linear, os indivíduos são ordenados de acordo com os valores de *fitness* e o nível N é associado ao melhor indivíduo e o nível 1, ao pior. Em seguida, a cada indivíduo i é associada uma probabilidade p_i de ser selecionado.

$$p_i = \frac{c-1}{c^{N-1}} c^{N-i} \quad \text{onde } i \in \{1,2,...N\} \qquad (7)$$

2.8 Operadores Genéticos

Uma vez que os indivíduos tenham sido selecionados, deve-se aplicar um dos operadores genéticos. Os três operadores principais são [Koza 1992]:

1) **Reprodução:** um programa é selecionado e copiado para a próxima geração sem sofrer nenhuma mudança em sua estrutura.

2) **Cruzamento** *(crossover)*: dois programas são selecionados e são recombinados para gerar outros dois programas. Um ponto aleatório de cruzamento é escolhido em cada programa-pai e as árvores abaixo destes pontos são trocadas. Um exemplo de cruzamento pode ser visto na Figura 7. Neste exemplo, foram escolhidos os programas: `((2*(x+x))+1)` e `(((x+1)*x)-2)`. Foram escolhidos aleatoriamente um nó em cada árvore, identificado com um traçado mais denso na figura. As árvores são então trocadas, gerando os novos programas: `((x+1)+1)` e `(2*((x+x)*x)-2)`.

Para que o cruzamento seja sempre possível, o conjunto de funções deve apresentar a propriedade de Fechamento (*closure*), isto é, as funções devem suportar como argumento qualquer outra função ou terminal. Se não for possível, devem-se estabelecer critérios de restrição na escolha dos pontos de cruzamento.

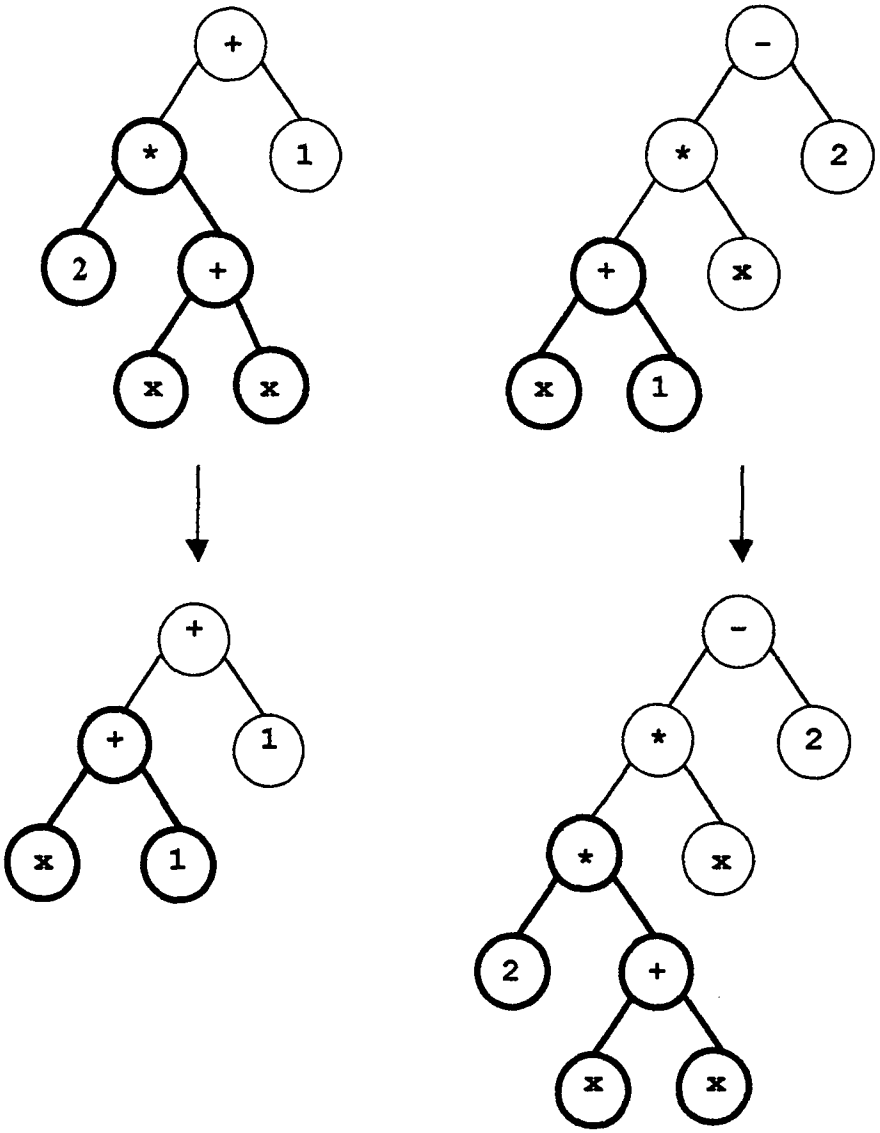

FIGURA 7: EXEMPLO DE CRUZAMENTO ENTRE DOIS PROGRAMAS

3) **Mutação** (*mutation*): um programa é selecionado e um de seus nós é escolhido aleatoriamente. A árvore cuja raiz é o nó selecionado é então eliminada e substituída por uma nova árvore gerada aleatoriamente.

2.9 Critério de Término

É responsável por interromper o laço de repetição do processo evolutivo que, idealmente, não teria fim. O critério mais comum é limitar o número máximo de gerações ou até que uma solução satisfatória seja encontrada [Koza 1992], porém existem critérios baseados no próprio acompanhamento do processo evolutivo, isto é, enquanto houver melhoria na média da população, o processo evolutivo prossegue [Kramer 2000]

2.10 Limitações

A obrigatoriedade da propriedade de fechamento (*closure*) limita os domínios a serem usados, não possibilitando a aplicação ampla da Programação Genética. A necessidade do fechamento é devida ao uso irrestrito dos operadores genéticos nos programas. Para contornar estes problemas, John Koza[8] [Koza 1992] propôs alterações que devem ser feitas no algoritmo para adequá-lo a domínios que apresentem restrições sintáticas. Porém, estas alterações direcionam o algoritmo para solucionar problemas de determinado tipo, restringindo a sua aplicabilidade.

Por exemplo, para encontrar uma solução em forma de Série de Fourier, John Koza propôs a adoção de três restrições [Koza 1992]:

- O nó-raiz da árvore deve ser obrigatoriamente a função especial &.
- As únicas funções possíveis abaixo de uma & são as funções trigonométricas `xsin`, `xcos` e &.
- As únicas funções permitidas abaixo das trigonométricas são as funções aritméticas (+, -, *, %[9]) ou uma constante.

[8] Capítulo 19 – *Evolution of Constrained Syntatic Structures*, pág. 479 a 526.
[9] Referindo-se a função de divisão protegida, isto é, % é idêntica a / com exceção de que uma divisão por zero resulta em um [Koza 1992]

A função especial & faz o mesmo que a função soma (+). A função `xsin` é definida como tendo dois argumentos (`arg0` e `arg1`) e seu cálculo é `arg1*sin(arg2*x)`, sendo que o valor de `arg2` é arredondado para o inteiro mais próximo. De forma semelhante, define-se a função `xcos`.

Estas restrições tornam-se necessárias pois uma Série de Fourier tem a forma: $a_0 + \sum_{i=1}^{\infty} (a_i \cos\theta + b_i sin\theta)$

Estas restrições são mantidas através da identificação do tipo de cada nó, evitando que os operadores genéticos violem o formato pré-determinado. Apesar deste método funcionar adequadamente para as Séries de Fourier, ele representa uma abordagem que exige a adequação do algoritmo para cada problema que apresente restrições em termos de formato de solução.

Para permitir a aplicabilidade da Programação Genética a qualquer problema que imponha restrições sintáticas, Frederic Gruau [Gruau 1996] propôs o uso de gramáticas. Ao invés de simplesmente informar os conjuntos F e T, as regras de formação dos programas também são fornecidas. Desta forma é possível guiar genericamente a aplicação dos operadores genéticos a fim de produzir programas sintaticamente corretos frente ao domínio. Na Seção 5.4 demonstra-se o uso de gramáticas para solucionar as Séries de Fourier.

No capítulo a seguir, as alterações necessárias para permitir o uso de gramáticas em Programação Genética são apresentadas.

3. PROGRAMAÇÃO GENÉTICA ORIENTADA A GRAMÁTICAS

Neste capítulo apresenta-se as alterações necessárias para adequar o algoritmo da Programação Genética a problemas descritos através de uma gramática[10]. Inicialmente, são apresentadas as vantagens do uso de gramáticas em Programação Genética. Em seguida, as mudanças na representação da população são descritas. As modificações necessárias no processo de criação da população e na atuação dos operadores genéticos são discutidas no decorrer do texto.

3.1 Motivação

Tradicionalmente, a linguagem alvo usada para Programação Genética é o LISP [Banzhaf 1998]. Graças a sua sintaxe simples e ao fato de tanto dados como programas terem o mesmo formato (*S-expressions*), tornou-se a linguagem "ideal" para evoluir programas [Angeline 1994]. As *S-expressions* podem ser:

- Um átomo, isto é, um símbolo (variável, função etc.) ou um não-símbolo (número, cadeia de caracteres etc.);

- Uma lista, composta por símbolos ou não-símbolos, no formato `( S-expr S-expr ...)`, por exemplo, `(+ 1 2)`, `(A (B C ))`;

A avaliação de uma *S-expression* é simples:

- Para avaliar um átomo, se ele é um não-símbolo, a sua avaliação é o próprio não-símbolo. Caso seja uma variável, a sua avaliação é o valor atual da variável;

- Para avaliar uma lista, assume-se que o primeiro elemento é uma função, sendo os elementos seguintes, seus argumentos.

[10] No Anexo A apresentamos um resumo sobre gramáticas para facilitar o entendimento deste capítulo.

Portanto é possível tratar os programas diretamente como se fossem dados, facilitando a geração e a aplicação dos operadores genéticos. Desta forma, a especificação de funções e terminais é suficiente para evoluir programas.

Porém, as diversas linguagens de programação, tais como C, Pascal ou Prolog, impõem restrições de formato e tipo, tornando muito difícil a aplicação da Programação Genética diretamente nestas linguagens. Além disto, há domínios em que restrições, quanto ao formato ou tipo, devem ser satisfeitas, tais como as Séries de Fourier apresentadas anteriormente.

Desta forma, a necessidade de se impor ou restringir o formato dos programas, seja por causa de sua linguagem ou por restrições de formato, exige mudanças no algoritmo de Programação Genética. Para cada tipo de linguagem ou restrição, há a necessidade de se adequar o algoritmo.

De forma a tornar a Programação Genética realmente genérica, isto é, aplicável a qualquer problema cuja solução possa ser em forma de um programa, Frederic Gruau [Gruau 1996] propôs o uso de gramáticas. A generalização se torna possível pelo fato de atualmente as linguagens de programação serem facilmente descritas por uma gramática.

Para o uso de gramáticas em Programação Genética, tornam-se necessárias modificações na forma de criação da população inicial e na atuação dos operadores genéticos. Estas alterações visam a preservação da consistência sintática dos programas durante o processo evolutivo e representam extensões dos originais com *S-expressions* [Whigham 1996].

As principais modificações com relação a Programação Genética tradicional são explicadas nas seções que seguem. Inicialmente, mostra-se a representação dos programas através de árvore de derivação. Posteriormente, o algoritmo de criação da população inicial é detalhado. As alterações necessárias ao comportamento dos operadores genéticos são apresentadas no final do capítulo.

3.2 Representação dos Programas

A representação dos programas criados a partir de uma gramática pode ser feita através de sua árvore de sintaxe concreta ou árvore de derivação [Whigham 1995; Ratle 2000]. Por exemplo, considere a gramática $G = \{N, \Sigma, S, P\}$ apresentada a seguir:

Σ = { x, +, -, *, / }
N = { <exp>, <op>, <var> }
S = <exp>
P = { <exp> → <var> |
(<exp> <op> <exp>)
<op> → + | - | * | /
<var> → x }

A árvore de derivação para uma das possíveis expressões deriváveis de G, tal como (x+(x*x)), está na Figura 8.

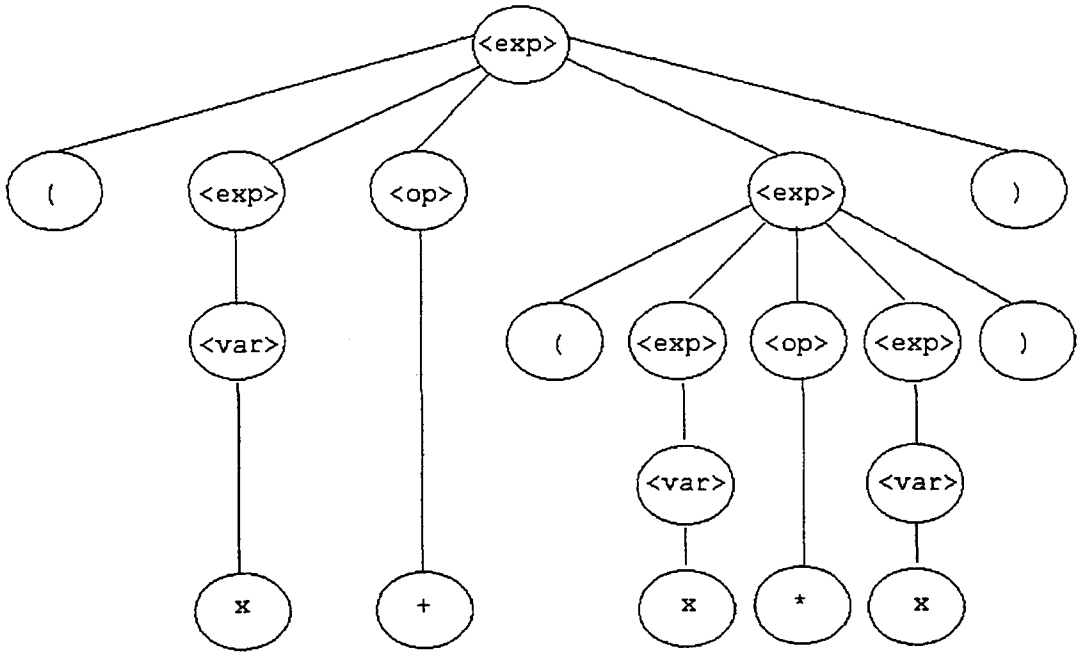

FIGURA 8: ÁRVORE DE DERIVAÇÃO PARA (X + (X *X))

3.3 População Inicial

O processo de criação de um programa baseado em uma gramática é simples. Partindo-se do símbolo inicial *S*, a árvore é construída através da aplicação sucessiva de produções a cada não-terminal, até que não seja mais possível aplicar nenhuma produção [Sethi 1996; Terry 1997]. Se os nós-folha neste instante forem compostos somente por terminais, diz-se que a árvore está *completa*. Uma profundidade máxima é geralmente especificada para evitar árvores muito grandes. Apesar de semelhante ao método *grow* [Koza 1992], não é possível, durante a geração da árvore, prever sua profundidade máxima. Somente após a árvore completa é possível avaliar sua profundidade. Isto leva a criação de um número expressivo de árvores que serão simplesmente rejeitadas.

Para evitar a rejeição de árvores recém criadas, Peter Whigham [Whigham 1996] propôs um método que se baseia em duas fases[11]. A primeira se preocupa em calcular o número mínimo de derivações de cada produção e a segunda, gera as árvores com base na profundidade desejada.

O algoritmo da primeira fase está na Figura 9.

Inicialmente, todas as produções tem MIN_DERIV, igual a -1.
Para cada produção na forma $A \rightarrow x_1\, x_2 \ldots x_n$ onde $x_1\, x_2 \ldots x_n \in \Sigma^*$
    Adota-se MIN_DERIV($A \rightarrow x_1\, x_2 \ldots x_n$) igual a zero(0).
    Enquanto houver uma produção $A \rightarrow \alpha$ com MIN_DERIV($A \rightarrow \alpha$) = -1
        Se todos os não-terminais $t_i \in \alpha$, apresentarem MIN_DERIV(t_i) ≠ -1
            MIN_DERIV($A \rightarrow \alpha$) = MAX(MIN_DERIV(t_i)) + 1.

FIGURA 9: ALGORITMO PARA O CÁLCULO DO NÚMERO MÍNIMO DE DERIVAÇÕES

[11] Este método também foi usado por Alain Ratle e Michele Sebag em [Ratle 2000]

Na segunda fase são criadas as árvores propriamente ditas através de um algoritmo semelhante ao *grow* [Koza 1992]. O algoritmo está na Figura 10.

Dados:
Profundidade máxima D
Conjunto de produções P
Símbolo inicial S
Faça:
Árvore = GROW(S, D)

GROW(símbolo inicial S, profundidade máxima d)
Retorna: uma árvore de derivação com profundidade máxima $\leq d$

Se $d = 1$
Retorne aleatoriamente uma produção $S \rightarrow x$ onde $x \in \Sigma^*$
Senão
Selecione aleatoriamente uma produção $S \rightarrow \alpha$ onde $\alpha \in \{N \cup \Sigma\}^*$ onde MIN_DERIV(S) $\leq d$
Para cada não-terminal $A \in \alpha$
Anexe a árvore retornada por GROW(A, d-1)

Retorne a árvore de derivação cuja raiz é o não-terminal S

FIGURA 10: ALGORITMO PARA CRIAÇÃO DA ÁRVORE DE DERIVAÇÃO

3.4 Operadores Genéticos

A atuação dos operadores de cruzamento e mutação devem respeitar a gramática usada, a fim de produzir programas válidos. Para o cruzamento, é necessário que os pontos de cruzamento sejam do mesmo tipo, isto é, pertencer ao mesmo não-terminal. Um exemplo de um cruzamento entre duas árvores de derivação está na Figura 11.

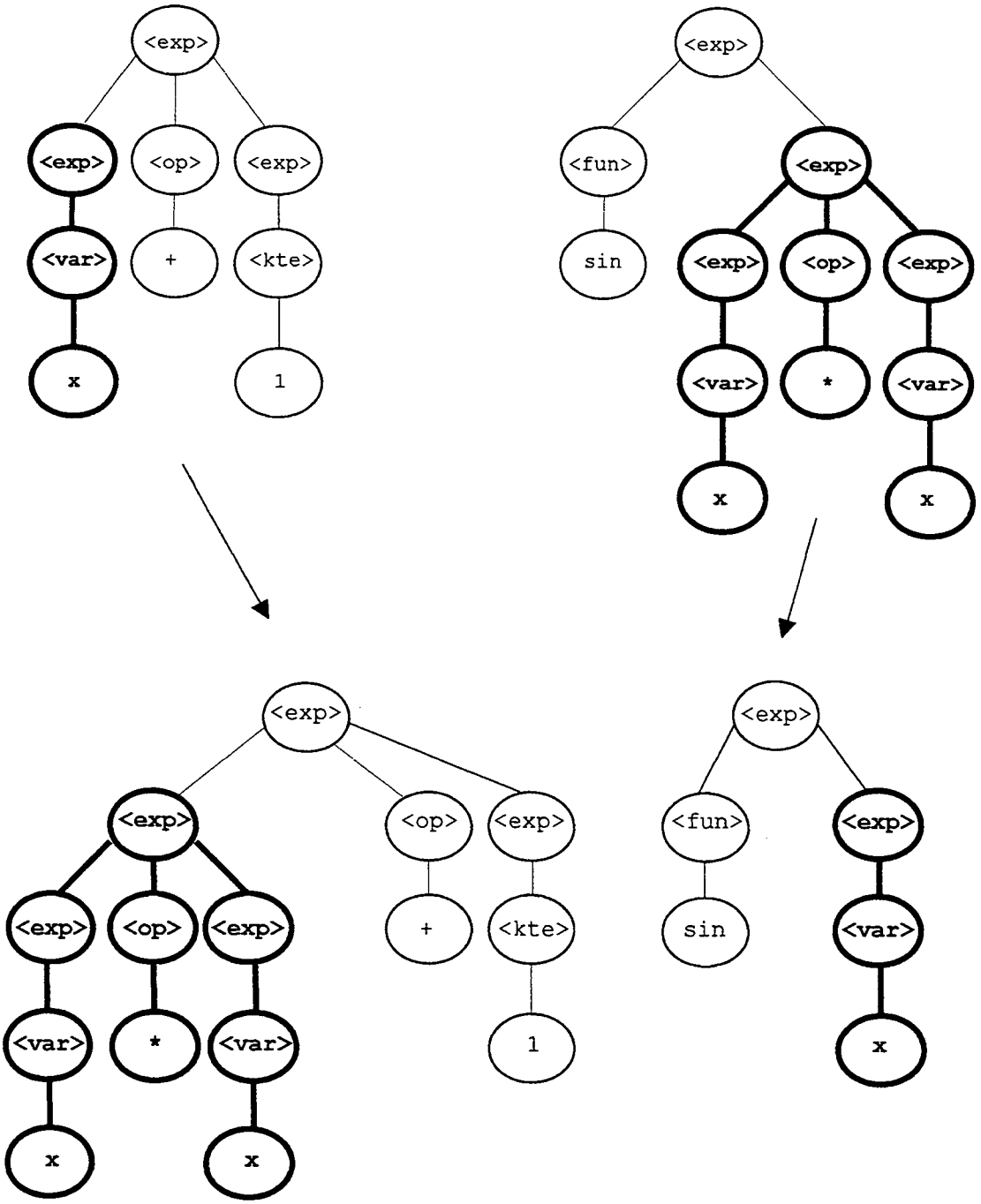

FIGURA 11: EXEMPLO DE CRUZAMENTO EM ÁRVORES DE DERIVAÇÃO[12]

[12] Para fins de legibilidade, os parêntesis foram propositavelmente omitidos.

O algoritmo para o operador de cruzamento está na Figura 12.

```
Dados:
        Duas árvores de derivação P1 e P2
        Profundidade máxima D
Faça:
        CROSSOVER(P1, P2, D)

CROSSOVER(árvore de derivação P1, árvore de derivação P2,
                profundidade máxima d)
      Retorna: duas árvores derivação com profundidade máxima ≤ d

        Selecione aleatoriamente um nó não-terminal A ∈ P1
        Construa uma lista de nós pertencentes a P2 cuja não-terminal seja A
        Se a lista resultar vazia
              Retorne com P1 e P2
        Senão
              Selecione aleatoriamente um nó da lista
              Permute as árvores de derivação abaixo de ambos os nós,
                 gerando dois novos programas P1' e P2'
              Se a profundidade de P1' é maior que d
                    Copie P1 em P1'
              Se a profundidade de P2' é maior que d
                    Copie P2 em P2'

        Retorne com P1' e P2'
```

FIGURA 12: ALGORITMO PARA CRUZAMENTO ENTRE ÁRVORES DE DERIVAÇÃO

Para o operador de mutação, o processo é semelhante. Após a escolha aleatória do ponto de mutação, substitui-se a subárvore abaixo dele por outra completamente nova, usando o mesmo algoritmo da população inicial.

3.5 Uso de Funções

A utilização de gramáticas em Programação Genética permite a geração de código em uma linguagem de programação diretamente e não somente expressões em LISP. Neste capítulo, as alterações necessárias para prover a Programação Genética do uso de gramáticas foram apresentadas.

Porém, as linguagens de programação permitem o uso de módulos ou funções, cujo objetivo principal é permitir a subdivisão de um problema maior em problemas menores [Terada 1991]. A Programação Genética, por lidar diretamente com a codificação de programas, não deve deixar de permitir a criação e uso de funções. O capítulo a seguir apresenta as diversas abordagens para evolução de funções que podem ser aplicadas a Programação Genética.

4. EVOLUÇÃO DE FUNÇÕES

Neste capítulo é feita uma visão geral dos principais métodos de evolução de funções em Programação Genética. Um detalhamento maior é dado ao método ADF e sua adaptação para o uso de gramáticas. Ao final, um exemplo do uso de ADF com gramáticas é apresentado.

4.1 Motivação

A Programação Genética não apresenta suporte de criação de módulos ou funções. Porém, em muitos casos as melhores soluções para problemas mais difíceis tendem a ser hierárquicas por natureza. A abordagem "dividir e conquistar" (*divide and conquer*) tem sido usada por humanos para lidar problemas complexos. A técnica se baseia na divisão do problema em problemas menores, mais fáceis de serem resolvidos, isto é, dada uma instância de um problema, de tamanho n, o método divide-a em k subinstâncias disjuntas ($1 < k \leq n$) que correspondem a k subproblemas distintos. Estes subproblemas são resolvidos separadamente e então acha-se uma forma de combinar suas soluções parciais para se obter a solução para a instância original [Terada 1991].

Várias extensões foram propostas para prover a Programação Genética da capacidade de evolução de funções, sendo as mais comuns [Yu 1999]: MA (*Module Acquisition*) [Angeline 1993], ADF (*Automatically Defined Functions*) [Koza 1994], ADM (*Automatically Defined Macros*) [Spector 1996] e ARL (*Adaptative Representation through Learning*) [Rosca 1996; Rosca 1997].

A seguir, cada uma destas extensões são descritas.

4.2 Module Acquisition (MA)

O uso de uma biblioteca de subrotinas é a base do método MA proposto por Peter Angeline [Angeline 1993]. Estas subrotinas podem ser chamadas por qualquer programa, quantas vezes for necessário.

A construção destas subrotinas é feita através de dois operadores novos: *Compressão* e *Expansão*. A Compressão cria subrotinas através da extração de uma "subárvore" de um programa. Esta "subárvore" é então substituída pelo nome da subrotina que é disponibilizada na biblioteca. A Expansão faz exatamente o inverso: ao encontrar em um programa a chamada à subrotina, a "subárvore" é incorporada ao programa.

Quando uma subrotina é criada inicialmente, apenas um programa a referencia. Se este programa apresentar um bom valor de *fitness*, ele será selecionado para a reprodução. Consequentemente, a "subárvore" que contém a chamada à subrotina poderá ser transferida para vários programas. Por outro lado, se o programa não apresentar um bom valor de *fitness*, menor será a probabilidade do uso da subrotina se estender a mais programas. Como resultado, subrotinas úteis são preservadas, facilitando o processo evolutivo.

As principais características de MA são [Kinnear Jr 1994]:

- As subrotinas, uma vez definidas, não evoluem;
- As subrotinas não podem ser recursivas, apesar de poderem chamar outras subrotinas (recursão indireta);
- As subrotinas contêm o mesmo conjunto de funções e terminais dos programas originais.

4.3 Automatically Defined Macros (ADM)

Lee Spector propôs o uso de ADM em Programação Genética [Spector 1995; Spector 1996] como uma forma de simultaneamente envolver programas e suas estruturas de controle. Quando o nome de uma ADM é chamada por um programa, uma substituição (*macro expansion*) é feita.

As ADMs são benéficas quando seus argumentos apresentam efeitos colaterais que são sensíveis ao ambiente em que são chamados. Por exemplo, no problema de movimentação de um robô (*obstacle-avoiding robot*), lida-se com programas que apresentam funções com efeitos colaterais tais como `LEFT` e `MOP` [Koza 1994]. Estas funções são sensíveis ao que está rodeando o robô e podem produzir resultados diferentes dependendo da sua posição. Em outras abordagens, estas funções fazem sempre a mesma coisa independentemente do contexto.

Experimentos feitos com o uso ADMs em movimentação de robôs têm apresentado resultados muito melhores que aqueles obtidos por outras abordagens [Spector 1996].

4.4 Adaptive Representation through Learning (ARL)

Semelhante a MA, o método ARL extrai segmentos de programas para criar funções [Rosca 1996]. Todavia, ao invés de adicioná-la a uma biblioteca, estas funções são adicionadas ao conjunto de funções F para serem usadas nas próximas gerações. Além disso, estas funções dinamicamente criadas podem ser excluídas do conjunto de funções se o seu uso não se demonstrar vantajoso. Consequentemente, o conjunto de funções expande e contrai durante o processo evolutivo.

A motivação para criação e destruição dinâmica de funções se baseia na reutilização de "bons" códigos de programa. Para atingir este objetivo, dois problemas devem ser resolvidos: (a) como reconhecer o que é um "bom" código e (b) quando adicionar ou excluir as funções. Em ARL, o reconhecimento é feito através da verificação da diferença de *fitness* entre pai e filho, buscando o "bloco" responsável pela melhoria. Para descobrir a necessidade de expansão ou contração do conjunto de

funções, uma medida global da entropia da população é feita para detectar se a busca está atingindo um ótimo local, sendo que uma modificação no conjunto de funções pode ser feita para "escapar" dele.

Porém, recentemente, Antonello Dessi [Dessi 1999] apresentou resultados que provam que ARL não mostra uma real capacidade de decomposição hierárquica de problemas, sendo que as soluções encontradas sempre exibem um baixo grau de modularidade e estrutura hierárquica.

4.5 Automatically Defined Functions (ADFs)

As ADFs representam um mecanismo proposto por John Koza para facilitar a criação e reutilização de módulos [Koza 1994]. Uma ADF é uma função que é desenvolvida durante o processo evolutivo e pode ser chamada pelo próprio programa ou outra função que esteja sendo desenvolvida simultaneamente. Ao solucionar problemas que apresentam regularidade em suas soluções, ADF fornece a Programação Genética um mecanismo para automaticamente decompor o problema em problemas menores e, então, usar a solução do problema menor para solucionar o problema. Além disso, Una-May O'Reilly demonstrou que ADFs também são vantajosas em outros métodos de busca, tais como o *Simulated Annealing* [O'Reilly 1996].

Uma ADF é uma função que pertence a um programa unicamente. Diferente de outras abordagens, as ADFs não são compartilhadas. Cada ADF tem o seu próprio conjunto de funções F e de terminais T. Inclusive, para promover melhor modularidade, é interessante que os conjuntos de funções das ADFs sejam agrupados por funcionalidade. Desta forma, a cada ADF é designada uma tarefa definida pelo seu conjunto de funções [Andre 1994].

Em seu trabalho, John Koza [Koza 1994] aplicou ADF a diversos problemas com sucesso: regressão simbólica, paridade par (*even parity*), cortador de grama (*lawnmower*), trilha da formiga (*ant trail*), campo minado (*minesweeper*), dentre outros.

4.6 ADF Orientada a Gramáticas

Na Programação Genética com ADF, os programas são compostos por uma parte principal (*main program*) e uma ou mais funções. Cada uma destas partes apresenta o seu próprio conjunto de funções e terminais [Koza 1994].

Uma abordagem possível para ADF orientada à gramáticas foi apresentada por Michael O'Neil [O'Neil 2000] com o nome de Definição de Funções Baseada em Gramática *(Grammar Based Function Definition)*, onde o código das ADF estende a gramática usada através do uso dos não-terminais `<adfcode>`, `<adfop>` etc. Nesta abordagem existe somente uma gramática, tornando-a mais complexa. A abordagem não foi aplicada em Programação Genética e sim em Evolução Gramatical[13] (*Grammatical Evolution*) [Ryan 1998]

Para tornar a geração das ADFs a mais próxima possível da proposta por John Koza [Koza 1994], adotou-se uma gramática para cada parte do programa, isto é, se existirem duas ADFs, deverão ser fornecidas três gramáticas (uma para o corpo principal e uma para cada ADF). Um detalhamento de como é feita a representação dos programas é feito a seguir. Em seguida, as modificações necessárias aos operadores genéticos são apresentadas.

Representação dos Programas

Cada programa é composto por uma ou mais árvores de derivação. A primeira árvore de derivação representa o corpo principal e cada uma das outras árvores, as funções ADF. Cada árvore segue uma gramática distinta. Como exemplo, considere que G é a gramática do corpo principal e G_0 a gramática para ADF0.

[13] *Grammatical Evolution* é uma algoritmo genético que manipula programas [Ryan 1998]

A gramática $G = \{\Sigma, N, S, P\}$ é definida como

```
Σ = { x, sin, cos, adf0 }
N = { <exp>, <func>, <var> }
S =  <exp>
P = { <exp>  →  <var> |
                fun(<exp>) |
                adf0(<exp>, <exp>)
      <fun>  →  sin | cos
      <var>  →  x }
```

A gramática $G_0 = \{\Sigma, N, S, P\}$ é definida como

```
Σ = { arg0, arg1, +, -, *, / }
N = { <exp>, <op>, <var> }
S =  <exp>
P = { <exp> →  <var> |
               (<exp> <op> <exp>)
      <op>  →  + | - | * | /
      <var> →  arg0 | arg1 }
```

Desta forma, um possível programa gerado é

Corpo principal: `sin(adf0(x,x))`

ADF0: `(arg0+arg1)`

Nas Figuras 13 e 14 mostram as árvores de derivação para este programa.

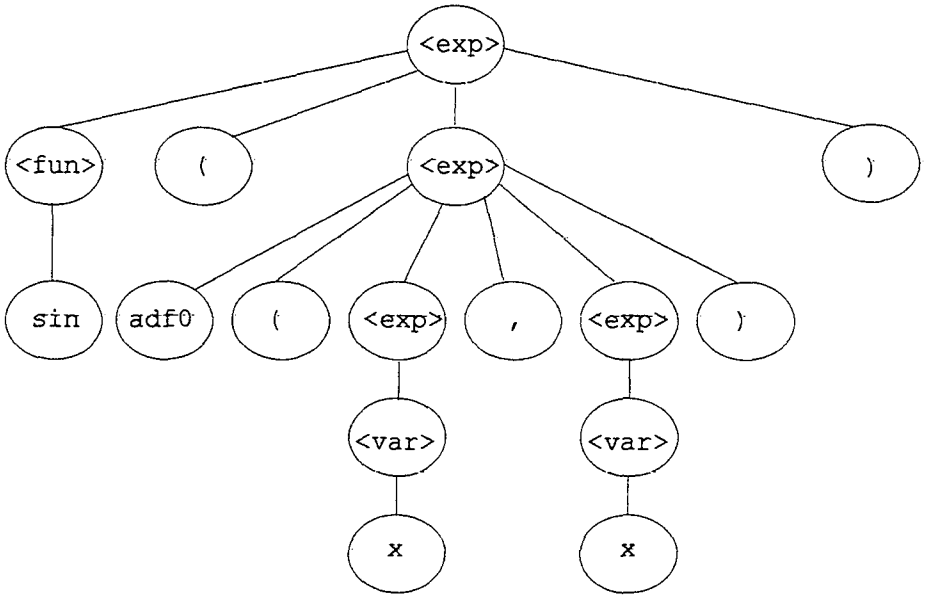

FIGURA 13: ÁRVORE DE DERIVAÇÃO PRINCIPAL PARA UM PROGRAMA COM UMA ADF

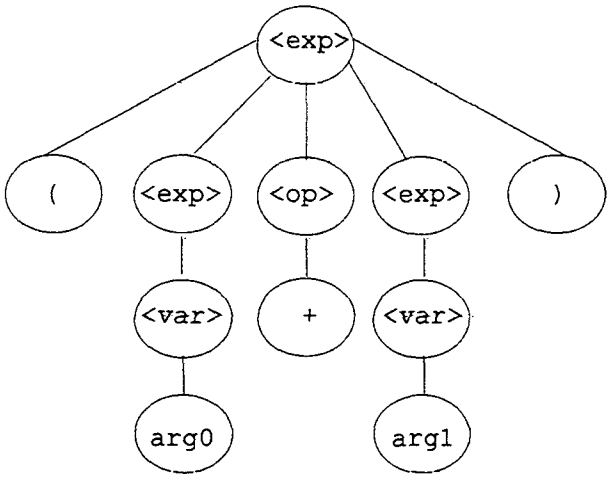

FIGURA 14: ÁRVORE DE DERIVAÇÃO DA ADF0 PARA UM PROGRAMA COM UMA ADF

Atuação dos Operadores Genéticos

Em Programação Genética tradicional, o uso de ADFs permite o uso de conjuntos diferentes de funções e terminais [Koza 1994]. Para evitar que durante a atuação de um operador genético, o programa use um terminal ou função que pertença a uma ADF, John Koza estabelece o uso de *branch typing*[14]. Basicamente, somente nós de um mesmo tipo de árvore podem ser trocados (cruzamento).

Da mesma forma que em Programação Genética tradicional, na orientada a gramáticas cada nó não-terminal é associado ao tipo de árvore a qual ele pertence. A partir do momento que um nó é escolhido aleatoriamente no primeiro pai, a escolha do nó no segundo pai é restrita aos nós cujo tipo de árvore seja idêntico ao do primeiro pai. Desta forma, as árvores de derivação geradas serão sempre consistentes com as suas respectivas gramáticas.

Após apresentar como aplicar o uso de gramáticas e evolução de funções em Programação Genética, é possível mostrar como foram implementadas na ferramenta. Isto é discutido no próximo capítulo.

[14] Seção 4.8 (*Structure-Preserving Crossover and Typing*) de [Koza 1994], pág. 85 a 87.

5. A FERRAMENTA CHAMELEON

Neste capítulo, uma ferramenta para Programação Genética Orientada à Gramáticas, chamada *Chameleon*, é descrita [Rodrigues 2001]. Chameleon foi desenvolvida em C++ em código portável, isto é, compilável tanto em ambiente Microsoft Windows® quanto em Linux[15]. O seu funcionamento e configuração serão abordados inicialmente. Os detalhes de sua implementação são apresentados no decorrer do texto. No final, apresenta-se um conjunto de exemplos de execução da ferramenta.

5.1 Estrutura Funcional

Para fins didáticos, é possível identificar três "atividades" principais em uma ferramenta de Programação Genética: (a) criação da população inicial, (b) avaliação dos programas (*fitness*) e (c) atuação dos operadores genéticos [Gritz, 1993]. Consequentemente, a funcionalidade básica de Chameleon pode ser facilmente desmembrada em três componentes principais: *Criador*, *Avaliador* e *Evolver*. A Figura 15 mostra a interligação destes componentes.

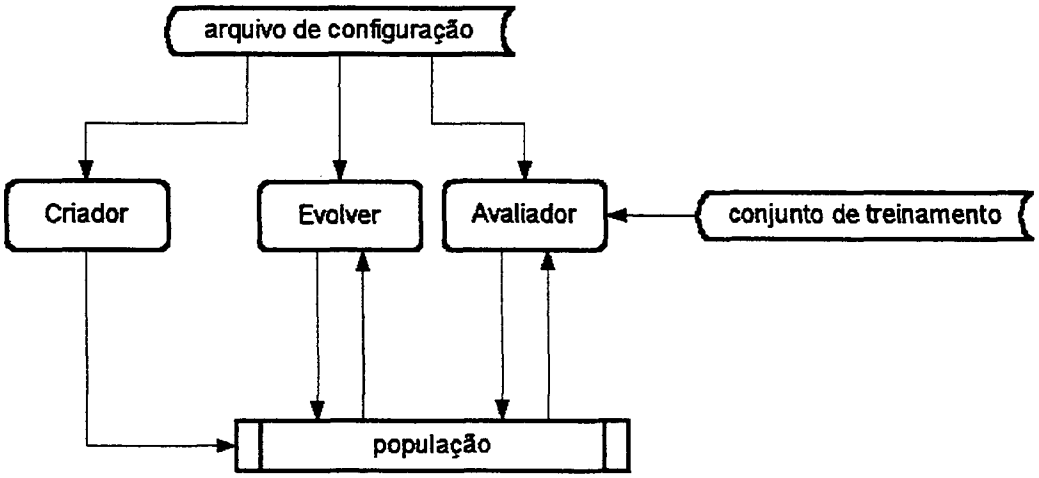

FIGURA 15: VISÃO FUNCIONAL DE CHAMELEON

[15] Plataformas testadas: Microsoft Windows (95, 98, NT 4, 2000) e Linux (Debian, Red Hat, Mandrake)

A.1.1 Criador

Este componente é responsável por duas tarefas: "compilar" a gramática e gerar a população inicial. A primeira se preocupa em converter a gramática informada em arquivo texto para uma representação interna, contendo todas as derivações possíveis das regras. Para cada não-terminal associa-se uma lista de derivações. Por exemplo, considere a gramática *G* apresentada a seguir:

Σ = { x, +, -, *, / }
N = { <exp>, <op>, <var> }
S = <exp>
P = { <exp> → <var> |
(<exp> <op> <exp>)
<op> → + | - | * | /
<var> → x }

Nesta gramática, existem três (3) não-terminais (<exp>, <op> e <var>) e sete (7) produções, duas (2) para o não-terminal <exp>, quatro (4) para <op> e uma (1) para <var>. Consequentemente, são criados três (3) nós, um para cada não-terminal, e a cada nó associa-se a lista de derivações. Na Figura 16, cada derivação é identificada por uma seta.

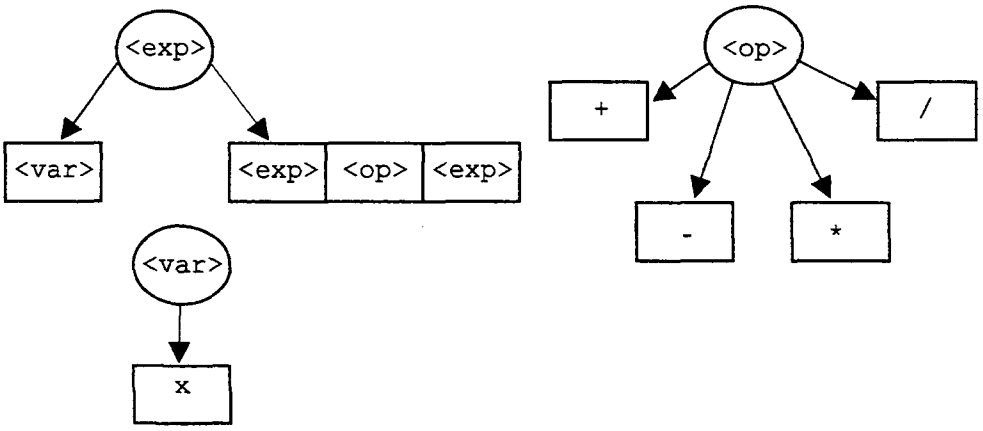

FIGURA 16: REPRESENTAÇÃO INTERNA DA GRAMÁTICA

Em seguida, a cada derivação de um não-terminal é associado um valor representando o número mínimo de passos para formar uma sentença contendo somente terminais. No exemplo apresentado, as derivações de <var> e <op> recebem o valor zero (0) pois todas as suas derivações resultam diretamente em terminal. A primeira derivação de <exp> recebe um (1) e a segunda, recebe dois (2). Estes valores são usados tanto pelo algoritmo de criação da população inicial quanto pelo de aplicação dos operadores genéticos para garantir que a profundidade máxima da árvore seja respeitada.

Após a montagem da estrutura interna da gramática, é possível gerar a população inicial conforme o algoritmo apresentado na Seção 3.3, isto é, a partir do não-terminal inicial (<exp>), escolhem-se sucessivamente de forma aleatória as produções cujas derivações não produzam uma árvore que ultrapasse a profundidade máxima desejada.

Avaliador

Os programas gerados por uma gramática podem pertencer a diferentes tipos de linguagens que exigem diferentes interpretações. Existem duas técnicas principais para avaliá-los [Whigham 1996]:

a) Avaliação direta das árvores: dada uma árvore de derivação tal que $S \Rightarrow^{+} x_1 x_2 ... x_n$, a árvore é convertida em uma árvore A de sintaxe abstrata. A avaliação de A é feita da seguinte forma: considerando que o nó $n \in A$ é o nó-raiz, então:

1. Se n é um terminal (variável ou constante), o seu valor é retornado.

2. Se n é uma função, então percorre-se cada um de seus nós-filho da esquerda para a direita. Considere cada nó-filho como n e vá para 1.

3. Após todos os nós-filho terem seus valores calculados, aplique a função do nó n (nó-pai).

A vantagem deste método é a rapidez da avaliação, porém há a necessidade de fornecer o interpretador da linguagem. Para este trabalho, desenvolveu-se um interpretador simples capaz de lidar com a grande maioria dos problemas propostos em [Koza 1992; Koza 1994].

b) **Execução externa:** isto é feito gerando um arquivo texto com as sentenças formadas pela árvore de derivação para, em seguida, executar um compilador externo. Após o fim da compilação, o arquivo executável resultante é executado. A desvantagem deste método é o tempo gasto para compilação de cada programa, porém habilita o uso de qualquer linguagem sem necessidade de alterações na ferramenta.

Após a avaliação da população, o Avaliador produz um resumo estatístico contendo os valores mínimo, máximo, média e desvio padrão para os valores de *fitness*, *hits*, tamanho e profundidade.

Evolver

Este componente aplica os operadores genéticos (cruzamento e mutação) nos programas selecionados com base no *fitness* para gerar novos. Os operadores têm sua ação restrita em função da gramática, isto é, a substituição de um nó é feita somente por outro pertencente ao mesmo não-terminal. Se for usado ADF, a restrição também se aplica a não-terminais pertencentes ao mesmo tipo de árvore. Desta forma, produz-se sempre programas sintaticamente corretos e estruturalmente consistentes.

Potencialmente, um operador genético pode ser aplicado em qualquer nó não-terminal da árvore de derivação. Em Chameleon, podem ser estabelecidas duas formas de restrição na escolha do nó, descritas a seguir.

- Com probabilidade de 90%, somente nós cuja produção não derive somente terminais podem ser escolhidos. Esta restrição simula a forma como é feita a escolha do ponto de cruzamento ou mutação em Programação Genética tradicional [Koza 1992];

- Estabelecimento de uma lista de não-terminais que podem ser escolhidos. Desta forma, é possível evitar a escolha de determinados nós;

Esta segunda restrição procura evitar comportamento diferente entre operadores genéticos tradicionais e orientados a gramáticas. Considere, como exemplo, a expressão `(x * x)` produzida pela gramática $G = \{ N, \Sigma, S, P \}$ onde

Σ = { x, +, -, *, / }
N = { <exp>, <op>, <var> }
S = <exp>
P = { <exp> → <var> |
(<exp> <op> <exp>)
<op> → + | - | * | /
<var> → x }

Uma possível árvore de derivação para `(x * x)` baseada em G está na Figura 17.

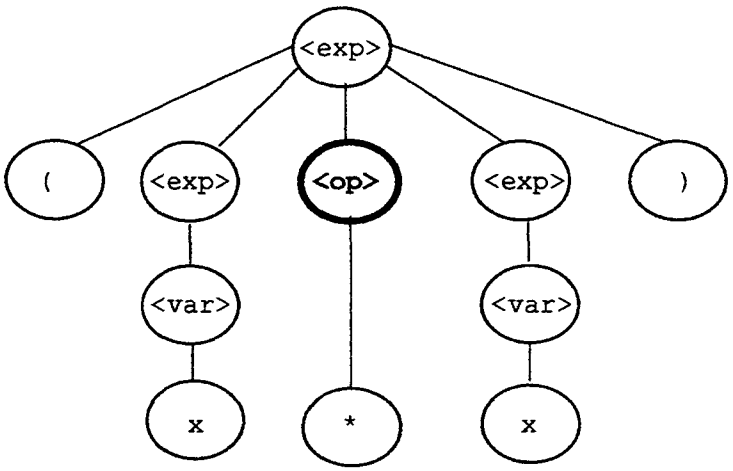

FIGURA 17: ESCOLHA DO NÃO-TERMINAL <OP>

O não-terminal <op> pode ser escolhido como ponto de cruzamento ou mutação. Porém, o único efeito que causaria na expressão final (sentença) seria a troca do operador "*" por outro. Em Programação Genética tradicional, um operador é considerado função e sempre tem nós abaixo dele, dada a sua representação em árvore de sintaxe abstrata. A sua escolha impacta, provavelmente, na alteração do operador e de seus argumentos também. Desta forma, o operador de cruzamento em Programação Genética orientada a gramáticas pode apresentar comportamento diferente em relação ao seu uso em Programação Genética tradicional [Whigham 1996; Ratle 2000].

Se for necessário evitar este comportamento diferente, pode-se restringir que o ponto de cruzamento ou mutação somente ocorra nos não-terminais <exp> e <var>.

5.2 Arquivo de Configuração

O arquivo de configuração estabelece os parâmetros e critérios a serem adotados durante a execução da ferramenta. O formato é texto, permitindo que qualquer editor possa criá-lo ou modificá-lo. O seu conteúdo é dividido em seções, onde cada seção é identificada por um nome envolvido pelos símbolos "[" e "]".

Os parâmetros fornecidos pelo arquivo de configuração são os valores iniciais adotados pela ferramenta. Se houver necessidade, a ferramenta permite que estes valores possam ser alterados durante sua execução, sem prejuízo de sua funcionalidade. Desta forma, é possível fazer a adequação de determinados parâmetros em função do andamento do processo evolutivo.

Todos os parâmetros necessários, exceto os *fitness cases*, são fornecidos por este arquivo, não havendo a necessidade de se especificar nenhum parâmetro na linha de comando além do nome do arquivo de configuração.

Qualquer linha existente antes da linha **[begin]** e após a linha **[end]** é ignorada, permitindo o uso de comentários, se necessário. Dentro das seções, a ordem das linhas é indiferente, isto é, "*tournament size*" pode vir antes de "*population size*". A validação de uma seção ocorre ao final dela e não durante sua leitura.

A seção **[parameters]** define os parâmetros principais. São eles:

- **population size:** número de programas que compõem uma população. Caso não seja informado, é assumido 1000.
- **selection method**: define o método de seleção a usar. Atualmente há suporte para dois tipos: 0 (*fitness proportionate*) e 1 (*tournament*). Caso não seja informado, é adotado 1 (*tournament*).
- **tournament size:** no caso do método ser por torneio (1), o tamanho do torneio é informado por este parâmetro. Caso não seja informado, adota-se 2.
- **minimum depth for initial random programs:** estabelece qual a profundidade mínima da árvore. Através deste parâmetro pode-se evitar geração de árvores muito pequenas na população inicial. Caso não seja informado, adota-se 2.
- **maximum depth for initial random programs:** estabelece a profundidade máxima possível da árvore na geração da população inicial. Caso não seja fornecido, adota-se 6.
- **avoid duplicates in initial random population:** define a exigência ou não de unicidade na população inicial. Se definido como "Y", todos os programas terão árvores de derivação distintas, mas não código distinto necessariamente. O valor padrão é "N".
- **avoid duplicates in the population:** define a exigência ou não de todas as populações geradas não apresentarem programas iguais. Se o valor for "Y", somente será aceito a inserção de um programa na população se este for único. O valor padrão é "N".
- **maximum depth for programs created during the run:** estabelece a profundidade máxima possível da árvore após aplicação dos operadores genéticos. Se não for informado, adota-se 17.
- **elitist:** se estiver com valor "Y", o melhor programa de cada geração será automaticamente copiado intacto para a geração seguinte. O valor padrão é "N".
- **threshold**: define qual a margem tolerada de erro para se considerar um acerto (*hit*) em cada *fitness case*. Caso não seja fornecido, adota-se 0.01
- **number of adfs:** define quantas ADFs deverão ser usadas. O valor padrão é 0, isto é, sem ADFs.

A seção **[result-producing branch]** define os parâmetros da gramática a ser usada no programa principal. É composto necessariamente pelas seguintes partes:

- **terminal set:** identifica o conjunto de terminais a serem usados.
- **function set:** identifica o conjunto de funções a serem usadas.
- **input variables:** identifica do conjunto de terminais quais são as variáveis. Na avaliação dos programas, cada variável recebe cada coluna de cada *fitness case* na ordem que foi declarada.
- **output variables:** identifica no código final gerado, qual ou quais variáveis terão as respostas geradas pelo programa.

A seção **[result-producing branch productions]** contém a notação BNF[16] das produções a serem aplicadas.

No caso de uso de ADFs, cada ADF tem as suas seções **[adf*n* branch]** e **[adf*n* branch productions]**. O padrão para os nomes das ADFs é idêntico ao adotado por [Koza 1994], isto é, a primeira ADF chama-se `ADF0`, a segunda, `ADF1` e assim por diante.

A seção **[crossover]** define a taxa e as restrições para o operador de cruzamento:

- **rate**: define a taxa de crossover (0 a 100). Se não informado, adota-se 90%. Se *rate = 0*, o operador não é utilizado.
- ***não-terminal* = *n***: define se *não-terminal* pode ser escolhido como ponto de cruzamento ou não. Se $n = 1$, então pode ser escolhido. Se $n = 0$, não será escolhido. No exemplo apresentado, o ponto de cruzamento só pode ser escolhido entre os não terminais `<exp>` e `<var>`.

[16] Para um esclarecimento sobre a notação BNF, leia Seção A.7 (pág. 99)

A seção **[mutation]** tem os mesmos critérios que a seção **[crossover].**

A seção **[fitness evaluation]** define como será a avaliação de *fitness*. Há dois casos possíveis dependendo do conteúdo da linha seguinte. Se o conteúdo for **interpreter**, um objeto CInterpreter será usado para avaliar os programas diretamente.

Para o uso do interpretador, os seguintes critérios devem ser seguidos:

- A gramática deve produzir programas cuja notação seja pré-fixada;
- As funções devem fazer parte da biblioteca (constar em `Biblio.h` e `Biblio.cpp`). Grande parte das funções usadas por John Koza [Koza 1992; Koza 1994] já estão implementadas[17]. A única execução é para o caso das ADFs, as quais são geradas automaticamente pela ferramenta;

Se o conteúdo for **compiler** = ***compilador***, será invocado *compilador* para avaliar os programas. Neste caso, as exigências são as seguintes:

- Deve existir um *wrapper* para permitir a inserção do código gerado pela ferramenta em um formato compilável;
- Os programas são gerados no diretório corrente com a extensão `PRG`
- O compilador é chamado via *system call*
- Em seguida, a ferramenta chama o executável `pop.exe`, portanto o compilador deve gerar necessariamente um executável com este nome.

A seção **[fitness cases]** define a localização e o nome do arquivo que contém os *fitness cases*. Isto é feito através da linha **source -> *nome_arquivo***. Caso não seja informada a localização (*path*), o diretório corrente é assumido.

O arquivo de configuração encerra com a linha **[end]**. Qualquer linha após é ignorada.

[17] No Anexo B tem-se uma listagem das funções implementadas em Chameleon.

5.3 Diagrama de Classes

Para o desenvolvimento da ferramenta, adotou-se a Programação Orientada a Objetos. O diagrama de classes em notação UML [Lee 1997] está na Figura 18.

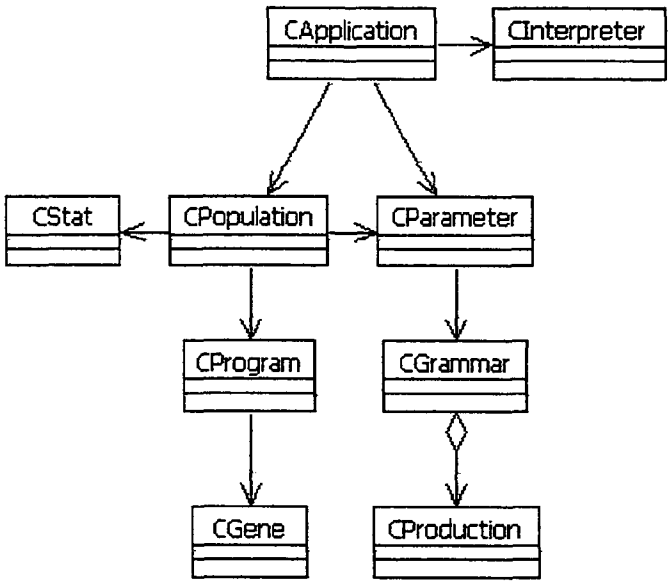

FIGURA 18: DIAGRAMA DE CLASSES DE CHAMELEON

No início da execução da ferramenta, um objeto da classe *CApplication* é criado. Toda a comunicação dos componentes básicos (Criador, Avaliador e Evolver) é feita através deste objeto. Todos os outros objetos necessários para execução são criados automaticamente por ele. Desta forma, ele representa uma aplicação ativa. Isto facilita a implementação de Programação Genética paralela, ou seja, basta criar mais de um objeto *CApplication* e coordenar suas execuções.

Após a criação do objeto da classe *CAplication*, dois métodos devem ser chamados: *Load* e *MakeGeneration*. O primeiro carrega os dados do arquivo de configuração e o segundo, permite que uma geração seja feita. O retorno do método *MakeGeneration* é um objeto *CProgram* que representa o melhor programa gerado.

Na chamada ao método *Load* de *CApplication*, um objeto *CParameter* é criado e preenchido com os parâmetros informados no arquivo de configuração. Da mesma forma, um objeto *CGrammar* também é criado para carregar a gramática. As regras de produção da gramática são armazenadas em uma lista STL (*Standard Template Library*) [Musser 2001] de objetos *CProduction*.

Durante a criação de um objeto *CApplication*, dois objetos são também criados da classe *CPopulation*: **parent** e **children**, representando a geração atual e a próxima, respectivamente. Cada um deles contem um conjunto de *n* objetos *CProgram*, onde *n* é o tamanho da população.

Cada *CProgram* tem uma lista contendo a seqüência de aplicação das regras de produção. Quando se torna necessário exibir o código fonte ou aplicar os operadores genéticos, a lista é expandida em um ou mais objetos *CGene*. Cada objeto *CGene* representa um nó da árvore de derivação do programa. O uso de uma lista com a seqüência de produções permite que cada programa ocupe o mínimo de memória possível, permitindo o uso de grandes populações sem prejuízo do desempenho.

As classes *CStat* e *CInterpreter* representam classes de apoio. Ao se criar um objeto *CStat*, ele deve ser associado a um objeto *CPopulation*. A partir desta associação, um levantamento estatístico da população é feito (cálculo da *fitness* média, desvio padrão da *fitness*, *fitness* mínimo e máximo, média de tamanho dos programas etc), permitindo avaliar a "qualidade" da população.

Um objeto *CInterpreter* permite avaliar os programas diretamente ou através de um compilador externo.

5.4 Exemplos de Execução com Uso de Interpretador

Problema sem restrição de formato

Para demonstrar o uso do interpretador em um problema sem restrição de formato, escolheu-se um problema clássico: *Regressão Simbólica*. Neste tipo de problema, o objetivo é encontrar uma expressão matemática que, dados os valores de entrada, produza o valor de saída desejado (ajuste de curvas). Considere, neste exemplo, a expressão $x^3 + x$ e o conjunto de dez *fitness cases* variando de zero (0) a nove (9) igualmente espaçados. Os valores dos *fitness cases* usados estão na Tabela 3.

	ENTRADA	SAÍDA
FITNESS CASE 1	0	0
FITNESS CASE 2	1	2
FITNESS CASE 3	2	10
FITNESS CASE 4	3	30
FITNESS CASE 5	4	68
FITNESS CASE 6	5	130
FITNESS CASE 7	6	222
FITNESS CASE 8	7	350
FITNESS CASE 9	8	520
FITNESS CASE 10	9	738

TABELA 3: CONJUNTO DE FITNESS CASES USADOS PARA REGRESSÃO SIMBÓLICA

O arquivo de configuração é composto pelo seguinte conteúdo:

1) Seção **[parameters]**

```
population size = 150
tournament size = 7
minimum depth for initial random programs = 3
maximum depth for initial random programs = 8
avoid duplicates in initial random population = Y
avoid duplicates in the population = N
maximum depth for programs created during the run = 19
elitist = Y
threshold = 0.01
number of adfs = 0
```

2) Seção **[result-producing branch]**

```
terminal set = {X}
function set = {SUM,SUB,MUL,DIV}
input variables = {X}
output variables = {Y}
```

3) Seção **[result-producing branch productions]**

```
<code> -> Y=<exp>;
<exp>  -> <var> | <fb>(<exp>,<exp>)
<fb>   -> SUM | SUB | MUL | DIV
<var>  -> X
```

4) Seção **[crossover]**

```
rate = 90
<code> = 0
<exp> = 1
<fb> = 1
<var> = 1
```

5) Seção **[mutation]**

```
rate = 0
```

6) Seção **[fitness evaluation]**

```
interpreter
```

7) Seção **[fitness cases]**

```
source -> teste.dat
```

Na Figura 19, a execução de Chameleon até o término da avaliação da geração 0 (zero) é exibida. Após a identificação da ferramenta, é informada a semente aleatória sendo usada. Em seguida, uma estatística da população é mostrada, seguida pelos dados do melhor programa da população.

```
*********************************************************
*  Chameleon - Grammar-Guided Genetic Programming System  *
*               Ernesto Luis Malta Rodrigues              *
*          version 2.0 ; Updated in November 2001         *
*********************************************************
random seed = 1015130749

Generation 0
Fitness..: Min = 48.9289 Max = 704187 Avg = 12913 Std = 74056.2
Hits.....: Min = 0 Max = 2 Avg = 0.59 Std = 0.587668
Size.....: Min = 1 Max = 63 Avg = 37.02 Std = 25.1741
Depth....: Min = 3 Max = 8 Avg = 7.12 Std = 1.28927
Fail.....: 0 programs
Population variety: 100%
Best of generation: Fitness: 48.9289 Hits: 0 Size: 25 Depth: 8
Execution time....: 1 s
Y=SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV(MUL(X,X),X)),DIV(MUL(X,DIV(X,X)),SUM(X,MUL
(X,X)))));
```

FIGURA 19: RESULTADO OBTIDO NA GERAÇÃO 0

Além da estatística sobre *fitness*, *hits*[18], tamanho e profundidade, a variedade da população (*population variety*) é mostrada. Ela representa o percentual de programas únicos na população, isto é, um valor de 100% significa que todos os programas são únicos. Este resultado já era esperado pois na seção `parameters` do arquivo de configuração, tem-se `avoid duplicates in initial random population` com o valor `Y`, conforme mostrado anteriormente.

É possível perceber que a população inicial é "ruim", pois a média de acerto (*hits*) é de apenas 0,59 com um desvio padrão de 0,587668. O melhor programa apresenta um *fitness* de 48,9289 e é equivalente a seguinte expressão aritmética:

```
X+((X*X*X-X*X/X)-(X*X/X)/(X+X*X))
```

Esta expressão simplificada resulta em $\frac{x^3 - x}{x^2 + x}$ que "vagamente se assemelha" à resposta correta $x^3 + x$. Na geração seguinte ocorre uma melhoria na "qualidade" da população, como pode ser visto na Figura 20.

[18] *Hits* representa a taxa de acerto, isto é, quantos *fitness cases* o programa acertou com base em uma margem de erro chamada *threshold* [Koza 1992]

```
Generation 1
Fitness..: Min = 45 Max = 11790 Avg = 1564.8 Std = 1217.63
Hits.....: Min = 0 Max = 2 Avg = 0.71 Std = 0.671121
Size.....: Min = 1 Max = 73 Avg = 34.26 Std = 26.0995
Depth....: Min = 3 Max = 12 Avg = 7.21 Std = 1.9607
Fail.....: 0 programs
Crossover rate....: 90% Mutation rate: 0%
Population variety: 51%
Lost of Diversity.: 68%
Best of generation: Fitness: 45 Hits: 1 Size: 5 Depth: 5
Execution time....: 1 s
Y=MUL(X,MUL(X,X));
```

FIGURA 20: RESULTADO OBTIDO NA GERAÇÃO 1

Na geração 1, a média de acertos (*hits*) aumentou de 0,59 para 0,71. O maior valor de *fitness* caiu significativamente de 704.187 para 11.790, sendo que o melhor programa apresenta uma taxa de acerto (*hits*) melhor (um contra zero da geração anterior) e um tamanho[19] muito menor (reduziu de 25 para apenas 5). A variedade da população também sofreu drástica alteração (caiu de 100% para apenas 51%).

Na estatística, aparecem novas medidas. A taxa real de cruzamento e de mutação são exibidas, representando efetivamente qual a parcela da população anterior em que foi aplicada. Aliado a variedade da população, tem-se a avaliação da perda de diversidade (*lost of diversity*). Esta avaliação representa o percentual da população anterior que deixou de ser selecionado. Nos torneios superiores a cinco (5), a perda de diversidade é superior a 50% [Blicke 1995].

Na seqüência, a Figura 21 apresenta os resultados das gerações seguintes até a solução.

[19] Tamanho de um programa é a quantidade de terminais e funções, também conhecido como complexidade estrutural (*structural complexity*) [Koza 1992]

```
Generation 2
Fitness..: Min = 6.78404 Max = 89922.9 Avg = 3447.52 Std = 11927.7
Hits.....: Min = 0 Max = 2 Avg = 0.49 Std = 0.541136
Size.....: Min = 1 Max = 83 Avg = 33.36 Std = 23.269
Depth....: Min = 3 Max = 13 Avg = 8.06 Std = 2.03415
Fail.....: 0 programs
Crossover rate....: 90% Mutation rate: 0%
Population variety: 53%
Lost of Diversity.: 66%
Best of generation: Fitness: 6.78404 Hits: 0 Size: 49 Depth: 12
Execution time....: 0 s
Y=SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV(MUL(X,X),SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV
(MUL(X,X),X)),DIV(MUL(X,DIV(X,X)),SUM(X,MUL(X,X)))))),DIV(MUL(X,DIV(X,X)),SUM
(X,MUL(X,X)))));


Generation 3
Fitness..: Min = 1.45002 Max = 136113 Avg = 4507.79 Std = 21245.4
Hits.....: Min = 0 Max = 3 Avg = 0.3 Std = 0.541229
Size.....: Min = 1 Max = 93 Avg = 31.02 Std = 22.6675
Depth....: Min = 3 Max = 15 Avg = 8.83 Std = 2.66687
Fail.....: 0 programs
Crossover rate....: 90% Mutation rate: 0%
Population variety: 54%
Lost of Diversity.: 63%
Best of generation: Fitness: 1.45002 Hits: 3 Size: 27 Depth: 11
Execution time....: 0 s
Y=SUM(X,SUB(MUL(X,MUL(X,X)),DIV(MUL(X,DIV(DIV(MUL(X,DIV(X,X)),SUM(X,X)),X)),SU
M(X,MUL(X,X)))));


Generation 4
Fitness..: Min = 1.41448 Max = 118799 Avg = 2810.31 Std = 16714.8
Hits.....: Min = 0 Max = 3 Avg = 0.37 Std = 0.733815
Size.....: Min = 1 Max = 133 Avg = 37.56 Std = 24.865
Depth....: Min = 3 Max = 17 Avg = 10.43 Std = 2.95848
Fail.....: 0 programs
Crossover rate....: 90% Mutation rate: 0%
Population variety: 53%
Lost of Diversity.: 66%
Best of generation: Fitness: 1.41448 Hits: 1 Size: 35 Depth: 11
Execution time....: 1 s
Y=SUM(X,SUB(MUL(X,MUL(X,X)),DIV(MUL(X,DIV(MUL(X,X),X)),SUM(X,SUB(MUL(MUL(X,X),
MUL(SUM(X,X),DIV(X,X))),DIV(MUL(X,X),X))))));


Generation 5
Fitness..: Min = 0 Max = 13302.9 Avg = 495.781 Std = 1936.75
Hits.....: Min = 0 Max = 10 Avg = 1.47 Std = 2.01737
Size.....: Min = 1 Max = 95 Avg = 33.72 Std = 18.2696
Depth....: Min = 3 Max = 17 Avg = 10.76 Std = 3.00545
Fail.....: 0 programs
Crossover rate....: 90% Mutation rate: 0%
Population variety: 38%
Lost of Diversity.: 68%
Best of generation: Fitness: 0 Hits: 10 Size: 7 Depth: 6
Execution time....: 0 s
Y=SUM(X,MUL(X,MUL(X,X)));


*** Solution found in generation 5 ***
Solution was saved in best.prg
Total execution time: 00:00:03 h
```

FIGURA 21: RESULTADOS OBTIDOS NAS GERAÇÕES SEGUINTES

As sucessivas respostas obtidas em cada geração com seus respectivos *fitness* estão na Tabela 4.

Geração	Solução Encontrada	Fitness
0	Y=SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV(MUL(X,X),X)),DIV(MUL(X,DIV(X,X)),SUM(X,MUL(X,X)))));	48.9289
1	Y=MUL(X,MUL(X,X));	45
2	Y=SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV(MUL(X,X),SUM(X,SUB(SUB(MUL(MUL(X,X),X),DIV(MUL(X,X),X)),DIV(MUL(X,DIV(X,X)),SUM(X,MUL(X,X))))))),DIV(MUL(X,DIV(X,X)),SUM(X,MUL(X,X)))));	6.78404
3	Y=SUM(X,SUB(MUL(X,MUL(X,X)),DIV(MUL(X,DIV(DIV(MUL(X,DIV(X,X)),SUM(X,X)),X)),SUM(X,MUL(X,X)))));	1.45002
4	Y=SUM(X,SUB(MUL(X,MUL(X,X)),DIV(MUL(X,DIV(MUL(X,X),X)),SUM(X,SUB(MUL(MUL(X,X),MUL(SUM(X,X),DIV(X,X))),DIV(MUL(X,X),X))))));	1.41448
5	Y=SUM(X,MUL(X,MUL(X,X)));	0

TABELA 4: SOLUÇÕES ENCONTRADAS PARA A REGRESSÃO SIMBÓLICA

Comportamento do Processo Evolutivo

Para ilustrar a evolução dos programas no exemplo apresentado, a Figura 22 mostra a evolução do valor *fitness* do melhor programa durante as gerações.

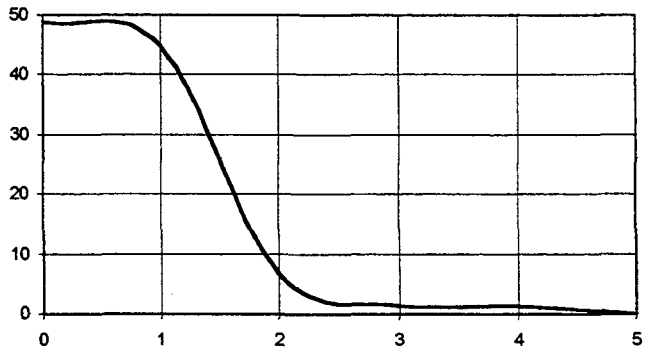

FIGURA 22: GRÁFICO DE FITNESS DO MELHOR PROGRAMA POR GERAÇÃO

Problema com restrição de formato

Uma das grandes limitações da aplicação da Programação Genética em qualquer problema reside na necessidade da propriedade de fechamento (*closure*). Através do uso de gramáticas é possível garantir a geração de programas válidos sintaticamente, não havendo a necessidade de satisfazer tal propriedade. Como exemplo de um problema com restrição de formato, tem-se a Série de Fourier, apresentada inicialmente na Seção 2.10. Nesta tipo de problema, o objetivo é encontrar uma determinada série que, dados os valores de entrada, produza o valor de saída desejado. Esta série deve respeitar o seguinte formato:

$$a_0 + \sum_{i=1}^{\infty} (a_i \cos\theta + b_i sin\theta)$$

Considere, neste exemplo, a expressão x^2 e o conjunto de cinqüenta *fitness cases* variando de $-\pi$ a π, igualmente espaçados. Os primeiros termos da Série de Fourier para esta expressão são:

$$x^2 = \frac{\pi^2}{3} - 4\cos x + \frac{4\cos 2x}{2^2} - \frac{4\cos 3x}{3^2} + \frac{4\cos 4x}{4^2} - \ldots$$

Neste problema, deve-se produzir programas com um número indefinido de termos, cada um contendo uma função seno ou coseno de um múltiplo de x. Não deve-se pré-determinar o número de termos a ser gerado, ou seja, o número de termos deve emergir do processo evolutivo.

O arquivo de configuração é composto pelo seguinte conteúdo:

1) Seção **[parameters]**

```
population size = 2000
tournament size = 7
minimum depth for initial random programs = 3
maximum depth for initial random programs = 8
avoid duplicates in initial random population = Y
avoid duplicates in the population = N
maximum depth for programs created during the run = 19
elitist = Y
threshold = 0.01
number of adfs = 0
```

2) Seção **[result-producing branch]**

```
terminal set = {X,ERC}
function set = {SUM,SUB,MUL,DIV,XSIN,XCOS}
input variables = {X}
output variables = {Y}
```

3) Seção **[result-producing branch productions]**

```
<code> -> Y=SUM(<trig>,<trig>);
<trig> -> SUM(<trig>,<trig>)
<trig> -> XSIN(<term>,<term>,X) | XCOS(<term>,<term>,X)
<term> -> ERC
<term> -> SUM(<term>,<term>) | SUB(<term>,<term>)
<term> -> MUL(<term>,<term>) | DIV(<term>,<term>)
```

4) Seção **[crossover]**

```
rate = 90
<code> = 0
<trig> = 1
<term> = 1
```

5) Seção ***[mutation]***

```
rate = 0
```

6) Seção **[fitness evaluation]**

```
interpreter
```

7) Seção **[fitness cases]**

```
source -> fourier.dat
```

Na Figura 23 é apresentada a execução de Chameleon até o término da avaliação da geração 0 (zero). Da mesma forma que o exemplo anterior, após a identificação da ferramenta, é informada a semente aleatória sendo usada. Em seguida, uma estatística da população é exibida, seguido pelos dados do melhor indivíduo da população.

```
*********************************************************
*  Chameleon - Grammar-Guided Genetic Programming System  *
*              Ernesto Luis Malta Rodrigues               *
*          version 2.0 ; Updated in November 2001         *
*********************************************************
random seed = 1015746538

Generation 0
Fitness..: Min = 62.4293 Max = 65464 Avg = 798.319 Std = 1918.55
Hits.....: Min = 0 Max = 2 Avg = 0.3055 Std = 0.710108
Size.....: Min = 4 Max = 128 Avg = 33.3962 Std = 35.6089
Depth....: Min = 3 Max = 8 Avg = 5.49717 Std = 1.70816
Fail.....: 0 programs
Population variety: 100%
Best of generation: Fitness: 62.4293 Hits: 2 Size: 32 Depth: 7
Execution time....: 67 s
Y=SUM(XCOS(SUB(-3.581,SUM(SUB(SUM(-1.427, 7.799),SUB( 6.563,-5.677)),SUB(SUB(
0.017, 8.824),MUL( 5.430, 2.888)))), 0.486,X),XCOS(SUM(SUM(SUB(MUL( 2.106,-
5.231),-1.211),SUB(SUM(-3.022, 1.905),SUM( 8.827,-5.200))),SUM(DIV(-9.358,DIV(
8.589,-5.096)),SUB(SUB(-9.986, 3.008), 0.324))),DIV(-6.233,SUB(SUB(SUM( 6.464,
7.356),SUB( 1.087,-7.339)),SUM(MUL( 3.571,-2.512),DIV( 7.444, 2.889)))),X));
```

FIGURA 23: RESULTADOS OBTIDOS NA GERAÇÃO 0

Devido ao fato da Série de Fourier ter um número indeterminado de termos, não é possível obter uma solução exata. Na Tabela 5 estão os valores de *fitness* do melhor programa, a média e o desvio padrão da população, a cada cinco gerações.

GERAÇÃO	FITNESS		
	MELHOR	MÉDIA	DESVIO PADRÃO
0	62.4293	798.319	1918.55
5	34.0403	956	44603.8
10	17.4578	92.569	338.705
15	16.7320	64.1511	224.405
20	15.7404	63.0639	239.187
25	15.7318	51.2687	142.957
30	15.7122	49.5746	136.673
35	15.6562	60.5766	302.728
40	15.6459	56.8086	527.347
45	15.6442	227.4	6943.58
50	15.3773	465.01	9834.06

TABELA 5: VALORES DE FITNESS A CADA CINCO GERAÇÕES

É possível perceber que nas dez primeiras gerações, o valor do melhor *fitness* cai de um pouco mais de 62 para um pouco mais de 17, representando que rapidamente se caminha para uma solução. Porém, a partir da 20ª. geração, o valor do melhor *fitness* se "congela" em torno de 15 a 16, não apresentando melhora significativa.

Através do desvio padrão, é possível perceber que até a 30ª. geração, há uma forte tendência de dominância de um grupo de valores de *fitness*, pois ocorre sua redução. Porém, entre a 20ª. e a 30ª. geração, apesar do desvio padrão reduzir-se, não há melhora na "qualidade" da população. É interessante notar que o algoritmo "sente" esta dominância e tende a "dispersar" os valores de *fitness* em seguida, dado o aumento significativo do desvio padrão após a 30ª. geração. Porém não "consegue" obter melhoria.

Esta característica de "congelamento" do valor de *fitness* deve-se a restrição imposta pela profundidade máxima da árvore de derivação, impedindo a geração de mais termos, dada que a série é infinita.

5.5 Exemplo de Execução com Uso de Compilador

Uma das grandes dificuldades da Programação Genética é a aplicação em domínios que envolvem estruturas de repetição. Isto se deve ao fato de ser praticamente impossível de se prever se um determinado programa apresenta ou não laço infinito, a não ser após o início de sua execução.

Um exemplo clássico é o Fatorial. Por definição, um fatorial de um número n é o resultado do produto de todos os números inteiros positivos até n. Por exemplo, se $n = 3$, o fatorial de n (denotado por $n!$) vale 1 x 2 x 3, ou seja, 6. Para $n = 4$, $n! = 1 \times 2 \times 3 \times 4 = 24$. Uma solução possível para um problema como este recai sobre o uso de estruturas de repetição, comuns em linguagens de programação.

Chameleon possui um mecanismo multitarefa para evolução de programas, isto é, a execução de um programa é feita sem a interrupção do núcleo. Desta forma, pode-se estabelecer um tempo máximo de execução (*timeout*). Se este tempo for

ultrapassado, o núcleo encerra a execução e associa o pior valor de *fitness* ao programa[20].

O conjunto de *fitness cases* usados para o fatorial está na Tabela 6.

	ENTRADA	SAÍDA
FITNESS CASE 1	0	1
FITNESS CASE 2	1	1
FITNESS CASE 3	2	2
FITNESS CASE 4	3	6
FITNESS CASE 5	4	24
FITNESS CASE 6	5	120
FITNESS CASE 7	6	720
FITNESS CASE 8	7	5040
FITNESS CASE 9	8	40320
FITNESS CASE 10	9	362880

TABELA 6: CONJUNTO DE FITNESS CASES USADOS PARA FATORIAL

Em seguida, o conteúdo do arquivo de configuração é apresentado.

1) Seção **[parameters]**

```
population size = 500
tournament size = 7
minimum depth for initial random programs = 5
maximum depth for initial random programs = 9
avoid duplicates in initial random population = Y
avoid duplicates in the population = Y
maximum depth for programs created during the run = 17
elitist = Y
threshold = 0.01
number of adfs = 0
```

2) Seção **[result-producing branch]**

```
terminal set = {x}
function set = {+, -, *, <, >, while}
input variables = {x}
output variables = {y}
```

[20] Uma outra alternativa seria eliminar este programa da população.

3) Seção **[result-producing branch productions]**

```
<code>  -> <stmt> while(<bool>) {<stmts>}
<stmt>  -> <var> = <exp>;
<stmts> -> <stmt> | <stmt> <stmts>
<exp>   -> <kte> | <var> <opa> <kte> | <var> <opa> <var>
<bool>  -> <var> <opl> <kte>
<opl>   -> < | >
<kte>   -> 0 | 1
<opa>   -> + | - | *
<var>   -> x | y
```

4) Seção **[crossover]**

```
rate = 90
<code> = 0
<stmt> = 1
<stmts> = 0
<exp> = 1
<bool> = 1
<opl> = 1
<kte> = 1
<opa> = 1
<var> = 1
```

5) Seção **[mutation]**

```
rate = 0
```

6) Seção **[fitness evaluation]**

```
compiler = cl -nologo -GX -Fepop.exe
```
[usando Microsoft Visual C++]

7) Seção **[fitness cases]**

```
source -> fatorial.dat
```

Na Figura 24 tem-se o início da execução de Chameleon até o término da avaliação da geração 0 (zero). Após a identificação da ferramenta, é informada a semente aleatória sendo usada. Em seguida, uma estatística da população é exibida, seguido pelos dados do melhor indivíduo da população.

```
*********************************************************
*  Chameleon - Grammar-Guided Genetic Programming System  *
*              Ernesto Luis Malta Rodrigues               *
*          version 2.0 ; Updated in November 2001         *
*********************************************************
random seed = 1015244063

Generation 0
Fitness..: Min = 792 Max = 1402 Avg = 872.644 Std = 29.6286
Hits.....: Min = 0 Max = 2 Avg = 0.339726 Std = 0.726035
Size.....: Min = 1 Max = 9 Avg = 3.46301 Std = 1.53776
Depth....: Min = 5 Max = 9 Avg = 7.456 Std = 1.38766
Fail.....: 135 programs
Population variety: 100%
Best of generation: Fitness: 792 Hits: 2 Size: 16 Depth: 7
Execution time....: 206 s
y=y+x; while(y<1) {y=x*x;x=1;x=x*1;}
```

FIGURA 24: RESULTADO OBTIDO NA GERAÇÃO 0

Neste problema, houve 135 programas que falharam. Provavelmente são os programas que causaram laço infinito ou cuja cálculo extrapolou o limite máximo de representação numérica (*overflow*). Outro aspecto interessante é o tempo de execução (206 s) representando mais de três minutos. Isto se deve ao tempo de compilação externa e ao número de programas (500).

A solução somente foi encontrada na 10ª. geração, mostrada na Figura 25.

```
Generation 10
Fitness..: Min = 0 Max = 895 Avg = 821.294 Std = 211.747
Hits.....: Min = 0 Max = 7 Avg = 1.17647 Std = 1.77607
Size.....: Min = 1 Max = 17 Avg = 10.4456 Std = 2.7895
Depth....: Min = 5 Max = 9 Avg = 7.456 Std = 1.38766
Fail.....: 160 programs
Population variety: 72%
Lost of diversity.: 68%
Best of generation: Fitness: 0 Hits: 10 Size: 13 Depth: 7
Execution time....: 316 s
y=1; while(x>0) {y=x*y; x=x-1; x=x*1;}
```

FIGURA 25: SOLUÇÃO ENCONTRADA NA GERAÇÃO 10

5.6 Aplicabilidade de Chameleon

Neste capítulo, o funcionamento da ferramenta Chameleon foi apresentado em detalhes. Para demonstrar seu funcionamento, três problemas foram usados: um de regressão simbólica, um de Série de Fourier (uso de restrição de formato) e o fatorial (uso de estrutura de repetição). Os passos para se configurar Chameleon foram suficientemente detalhados. Uma rápida discussão dos resultados apresentados foi feita. O objetivo principal foi demonstrar o funcionamento da ferramenta e não analisar detalhadamente os resultados.

Basicamente, as principais características de Chameleon são:

1. **Uso de gramática**: a descrição do formato do programa é feita através de uma gramática em notação BNF. Desta forma, Chameleon pode evoluir qualquer "indivíduo" que possa ser descrito por uma gramática;

2. **Critério de escolha da solução baseado em número de acertos**: para cada *fitness case* cuja diferença absoluta entre o resultado do programa e o correto for inferior a um limiar (*threshold*), Chameleon considera como um acerto (*hit*). Se o número de acertos de um programa for igual ao número de *fitness cases*, ele é considerado uma solução e o processo evolutivo termina. Se houver mais de um programa nestas condições, o que tiver menor tamanho é o escolhido;

3. **Avaliação de *fitness* baseada em erro**: Chameleon considera o *fitness* como sendo a soma absoluta das diferenças entre o resultado gerado pelo programa e o resultado correto. Desta forma, Chameleon usa aptidão padronizada (*standardized fitness*) onde quanto menor o valor, melhor o programa;

4. **Avaliação de *fitness* pode ser interna ou externa**: a avaliação de *fitness* dos programas pode ser feita internamente (através de um interpretador anexado à ferramenta) ou externamente (chamando um compilador externo), permitindo uma maior flexibilidade no uso da ferramenta;

5. **Baixo consumo de memória**: cada programa é armazenado como uma lista das produções usadas para produzi-lo e não através de sua árvore. Quando há necessidade de avaliação ou aplicação dos operadores genéticos, a sua árvore é montada. Desta forma, uma economia substancial de memória é feita em comparação a outras ferramentas que armazenam as árvores diretamente;

6. **Gravação e Recuperação de populações**: Chameleon permite que se grave em disco o conteúdo de uma população para recuperação posterior;

7. **Controle de código anômalo**: Chameleon controla o tempo de execução de cada programa, evitando programas que apresentem laço infinito. Caso ocorra, o programa é interrompido e penalizado com o pior valor de *fitness*.

8. **Suporte ao uso de ADF**: Chameleon permite o uso de ADF através da especificação da gramática de cada uma delas.

No próximo capítulo Chameleon será aplicado a um problema de maior complexidade: paridade par. O uso de evoluções de funções, através de ADFs, será abordado.

6. EXPERIMENTOS

Neste capítulo aborda-se o uso da ferramenta em um problema clássico da Programação Genética, o problema da paridade. A escolha se deve ao fato de vários pesquisadores o usarem como uma forma de *benchmark* para Programação Genética [Koza 1992; Koza 1994; Wong 1996; Chellapilla 1997; Gathercole 1997; Poli 1999; Yu 1999]. Além disso, foi um dos problemas escolhidos por John Koza para demonstrar o uso de ADF [Koza 1994], permitindo uma comparação direta entre a sua abordagem e a apresentada neste trabalho.

6.1 Metodologia

Diversos parâmetros de execução devem ser estabelecidos inicialmente tais como: tamanho da população, taxa de cruzamento etc. Para obter a maior fidelidade possível nas comparações, os parâmetros de execução usados foram os mesmos usados por John Koza [Koza 1994] e estão resumidos na Tabela 7.

Parâmetro	Valor
Tamanho da População	4.000 (com ADF) 16.000 (sem ADF)
Número Máximo de Gerações	50
Método de Avaliação de *fitness*	Aptidão Padronizada
Método de Seleção	Torneio de tamanho 7
Uso de Elitismo (reproduzir sempre o melhor)	Não
Probabilidade de Cruzamento	90%
Probabilidade de Mutação	0%
Probabilidade de Reprodução	10%
Probabilidade de escolha de pontos internos para cruzamento	90%
Profundidade Mínima dos programas na População Inicial	4
Profundidade Máxima dos programas na População Inicial	8
Profundidade Máxima permitida após cruzamento	19

TABELA 7: PARÂMETROS DE EXECUÇÃO PARA OS EXPERIMENTOS

Os valores de profundidade mínima e máxima dos programas na população inicial e a profundidade máxima permitida após o cruzamento diferem dos adotados por John Koza [Koza 1994]. Estes valores foram ajustados para apresentar semelhança comportamental, dado que as árvores de derivação são maiores que as árvores de sintaxe abstrata.

Desta forma, foi possível comparar diretamente os resultados e comprovar a viabilidade da abordagem apresentada. Ao final, uma comparação do uso ou não de ADF em Programação Genética orientada a gramáticas é feita, incluindo um teste de hipóteses.

As execuções foram feitas em plataforma Debian Linux em um Dual Pentium II de 500 MHz com 1 GB de memória.

6.2 Descrição do Problema

Os problemas de paridade são simples de definir, porém de difícil aprendizagem porque padrões semelhantes de entrada produzem respostas distintas [Franco 2001]. Além do mais, não apresentam nenhuma regularidade que possa ser descrita em forma de uma correlação entre os valores de entrada e saída.

Basicamente, os problemas de paridade envolvem um teste que determina se o número de uns ou zeros em uma seqüência de dígitos binários é par ou ímpar [Rumelhart 1986]. No caso da paridade-par (*even-parity*), a regra é a seguinte: a saída vale um se um número par de uns for fornecido como entrada. Por exemplo, considerando uma seqüência de três dígitos binários como entrada, a saída baseada em paridade-par (*even-3-parity*) é obtida conforme a Tabela 8.

ENTRADA	SAÍDA
0 0 0	1
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	0

TABELA 8: EVEN-3-PARITY FITNESS CASES

6.3 Especificação do Problema

Segundo John Koza, existem cinco passos principais na preparação de um problema: especificação do conjunto de terminais, especificação do conjunto de funções, forma de medida para o *fitness*, parâmetros para controle de execução e o critério de término. Estes passos são expressos através de uma tabela (*tableau*) [Koza 1992]. Na Tabela 9 estão os valores usados para os experimentos de paridade par sem uso de ADF [Koza 1994].

Objetivo	Obter um programa que produza a paridade par quando receber n valores como entrada.
Conjunto de terminais	n variáveis: D0, D1,..D_{n-1}
Conjunto de funções	AND, OR, NAND e NOR
Fitness cases	Todas as 2^n combinações dos n argumentos de entrada
Raw fitness	O número de fitness cases para o qual o valor retornado pelo programa é igual ao correto
Standardized fitness	A soma, por todos os 2^n fitness cases, do erro absoluto entre o valor retornado pelo programa e a resposta correta
Hits	O mesmo que *raw fitness*
Wrapper	Nenhum
Parâmetros	$M = 16{,}000$; $G = 51$
Detecção de Sucesso	O programa que atingir o máximo número de *hits*

TABELA 9: PARÂMETROS PARA O PARIDADE PAR SEM ADFS

A gramática usada para o caso de n = 3 (*even-3-parity*) está mostrada a seguir:

```
Σ = { AND, OR, NAND, NOR, D0, D1, D2 }
N = { <exp>, <fn>, <var> }
S =  <exp>
P = { <exp> → <var> |
              <fn>(<exp>, <exp>)
      <fn>  →  AND | OR | NAND | NOR
      <var> →  D0 | D1 | D2 }
```

Para os casos de $n > 3$, a única alteração na gramática é no conjunto de terminais. No caso de $n = 4$, os terminais são `D0`, `D1`, `D2` e `D3`. Para $n = 5$, inclui-se `D4`.

O conteúdo do arquivo de configuração usado para o paridade par com $n = 3$, sem ADF, está na Figura 23.

Para o caso de uso de ADF, é feito uma tabela adicional para especificar a composição dos programas conforme mostrado na Tabela 10.

Objetivo	Obter um programa que produza a paridade par quando receber n valores como entrada.
Arquitetura dos programas	Um programa principal (*result-producing branch*) e duas funções (*function defining branch*) com dois argumentos, com ADF1 hierarquicamente se referindo a ADF0
Parâmetros	*Branch typing*
Conjunto de terminais para o programa principal	n variáveis: D0, D1,..D_{n-1}
Conjunto de funções para o programa principal	AND, OR, NAND, NOR, ADF0 e ADF1
Conjunto de terminais para ADF0	n-1 argumentos:ARG0, ARG1,... ARG_{n-2}
Conjunto de funções para ADF0	AND, OR, NAND e NOR
Conjunto de terminais para ADF1	n-1 argumentos:ARG0, ARG1,... ARG_{n-2}
Conjunto de funções para ADF1	AND, OR, NAND, NOR e ADF0

TABELA 10: PARÂMETROS ADICIONAIS PARA O PARIDADE PAR COM ADFS

```
[begin]
[parameters]
population size = 16000
tournament size = 7
minimum depth for initial random programs = 4
maximum depth for initial random programs = 8
avoid duplicates in initial random population = Y
avoid duplicates in the population = N
maximum depth for programs created during the run = 19
elitist = N
threshold = 0.01
number of adfs = 0
[result-producing branch]
terminal set = {D0,D1,D2}
function set = {AND,OR,NAND,NOR}
input variables = {D0,D1,D2}
output variable = P
[result-producing branch productions]
<code> -> P=<exp>;
<exp>  -> <var> | <fb>(<exp>, <exp>)
<fb>   -> NAND | AND | NOR | OR
<var>  -> D0 | D1 | D2
[crossover]
rate = 90
<exp> = 1
<fb> = 0
<var> = 1
[mutation]
rate = 0
[fitness evaluation]
interpreter
[fitness cases]
source -> even3.dat
[end]
```

FIGURA 23: ARQUIVO DE CONFIGURAÇÃO PARA EVEN-3-PARITY SEM ADF

Quanto aos problemas de paridade par para $n > 3$, o arquivo de configuração é basicamente o mesmo, apenas alterando-se a quantidade de terminais usados.

O arquivo de configuração usado para o paridade par com $n = 3$ com ADF está na Figura 24. Tanto `ADF0` como `ADF1` apresentam aridade dois, sendo que `ADF1` pode chamar `ADF0`. No caso de $n > 3$, as aridades das ADF são corrigidas para n-1, isto é, `ADF0` e `ADF1` para paridade par com $n = 4$ devem apresentar aridade três [Koza 1994].

```
[begin]
[parameters]
population size = 4000
tournament size = 7
minimum depth for initial random programs = 4
maximum depth for initial random programs = 8
avoid duplicates in initial random population = Y
avoid duplicates in the population = N
maximum depth for programs created during the run = 19
elitist = N
threshold = 0.01
number of adfs = 2
[result-producing branch]
terminal set = {D0,D1,D2}
function set = {AND,OR,NAND,NOR,ADF0}
input variables = {D0,D1,D2}
output variables = {P}
[result-producing branch productions]
<code> -> P=<exp>;
<exp>  -> <var> | <fb>(<exp>, <exp>) | <adf>(<exp>,<exp>)
<fb>   -> NAND | AND | NOR | OR
<adf>  -> ADF0 | ADF1
<var>  -> D0 | D1 | D2
[adf0 branch]
terminal set = {ARG0,ARG1}
function set = {AND,OR,NAND,NOR}
input variables = {ARG0,ARG1}
output variables = {adf0}
[adf0 branch productions]
<code>  -> adf0=<exp>;
<exp>  -> <var> | <fb>(<exp>,<exp>)
<fb>   -> NAND | AND | NOR | OR
<var>  -> ARG0 | ARG1
[adf1 branch]
terminal set = {ARG0,ARG1}
function set = {AND,OR,NAND,NOR,ADF0}
input variables = {ARG0,ARG1}
output variables = {adf1}
[adf1 branch productions]
<code> -> adf1=<exp>;
<exp>  -> <var> | <fb>(<exp>, <exp>) | <adf>(<exp>,<exp>)
<fb>   -> NAND | AND | NOR | OR
<adf>  -> ADF0
<var>  -> ARG0 | ARG1
[crossover]
rate = 90
<exp> = 1
<fb> = 0
<var> = 1
[mutation]
rate = 0
[fitness evaluation]
interpreter
[fitness cases]
source -> even3.dat
[end]
```

FIGURA 24: ARQUIVO DE CONFIGURAÇÃO PARA EVEN-3-PARITY COM ADF

6.4 Exemplos de Soluções Encontradas

A Programação Genética é capaz de gerar soluções em forma de programas de computador, mas devido ao seu processo estocástico, cada execução gera uma solução de formato diferente, porém equivalente a correta. Muitas partes do código apresentam-se redundantes ou inúteis, porém a funcionalidade do programa é coerente com os *fitness cases* informados.

Por exemplo, uma solução encontrada para com $n = 3$ é mostrada a seguir. Em negrito identificamos duas partes que poderiam ser simplificadas.

```
P=OR(AND(NAND(AND(OR(D2,NAND(OR(D0,D0),D2)),OR(AND(D1,D1),OR(NAND(D2,
D1),D0))),AND(OR(NOR(AND(D1,D2),D0),NOR(D2,OR(AND(D1,D2),D1))),OR(D2,
OR(D0,OR(D2,D1))))),NAND(AND(D2,AND(D0,D1)),D2)),NOR(OR(D1,D0),D2));
```

O mesmo ocorre também com uso de ADF. Por exemplo, uma solução encontrada para $n = 4$ com ADF é mostrada a seguir. Em negrito identificamos duas partes que poderiam ser simplificadas.

```
P=ADF0(ADF0(NOR(ADF1(ADF0(NOR(AND(D0,NAND(D3,D2)),NAND(OR(D2,ADF1(NAND
(D1,D0),NAND(D3,D3))),ADF0(OR(D0,D3),ADF0(ADF1(D1,D1),D1)))),D0),NOR
(D2,D2)),NAND(NAND(D1,D1),OR(D2,D3))),D1),NOR(ADF1(ADF0(D1,D0),NOR(D2,
D2)),NAND(ADF0(D3,D3),NAND(NOR(D1,D3),D1)))));
adf0=OR(AND(ARG0,AND(ARG0,ARG1)),NOR(ARG0,ARG1));
adf1=NOR(OR(ARG0,ARG1),ADF0(ADF0(AND(ARG1,ARG1),AND(ARG1,ARG0)),AND(OR
(ARG0,ARG1),ARG1)));
```

6.5 Resultados sem Uso de ADF

Nas Figuras 25, 26 e 27 apresentamos os gráficos de Probabilidade de Sucesso, P(M,*i*) para o paridade par com n = 3, 4 e 5 respectivamente. Os resultados estão baseados em 34 (n = 3), 18 (n = 4) e 25 (n = 5) execuções independentes[21]. Uma aproximação dos gráficos apresentados em [Koza 1994] foi feita para prover uma comparação entre a ferramenta e a Programação Genética tradicional.

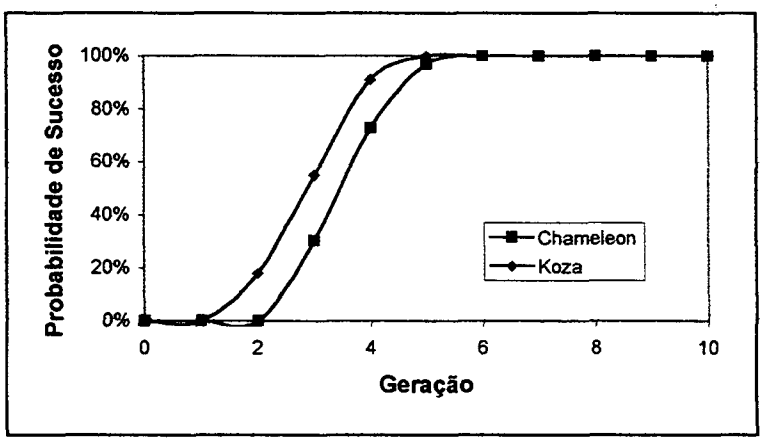

FIGURA 25: PROBABILIDADE DE SUCESSO PARA N = 3 SEM ADF

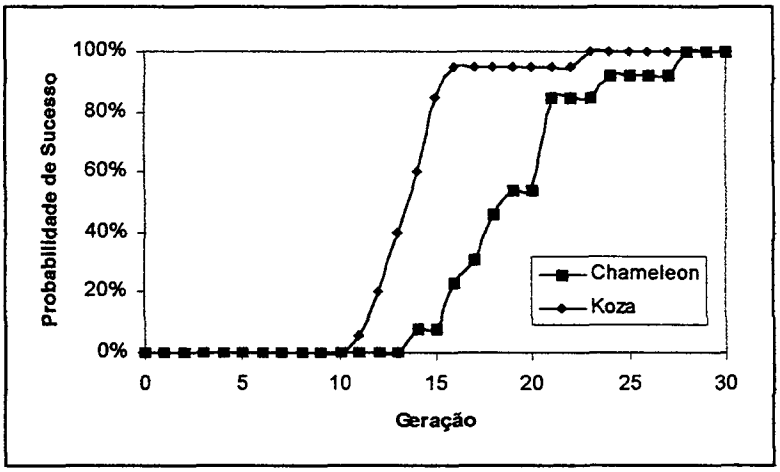

FIGURA 26: PROBABILIDADE DE SUCESSO PARA N = 4 SEM ADF

[21] Estes números de execuções correspondem aos usados por John Koza [Koza 1994]

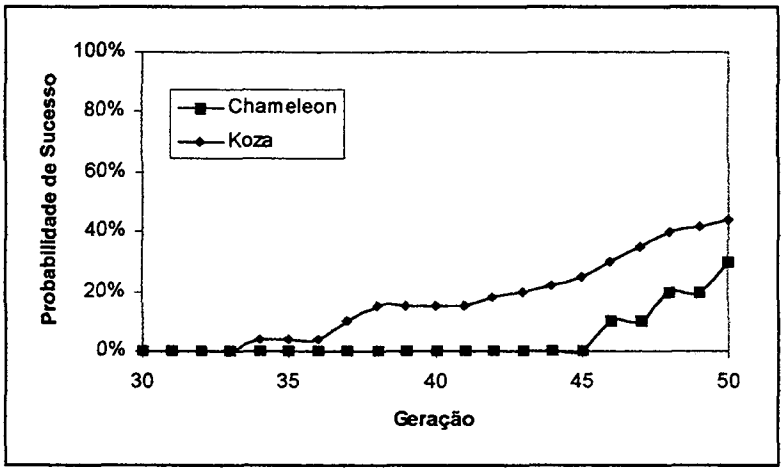

FIGURA 27: PROBABILIDADE DE SUCESSO PARA N = 5 SEM ADF

Pelos gráficos, percebe-se que a convergência da ferramenta é levemente mais lenta. Este comportamento pode estar associado a diferença de representação dos programas e a forma de aplicação do operador de cruzamento.

6.6 Resultados com Uso de ADF

Nas Figuras 28, 29 e 30 apresentamos os gráficos de Probabilidade de Sucesso, P(M,*i*) para o paridade par com n = 3, 4 e 5 respectivamente, usando ADF. Os resultados estão baseados em 33 (n = 3), 18 (n = 4) e 19 (n = 5) execuções independentes. Da mesma forma que a seção anterior, uma aproximação dos gráficos apresentados em [Koza 1994] foi feita para prover uma comparação entre a ferramenta e a Programação Genética tradicional.

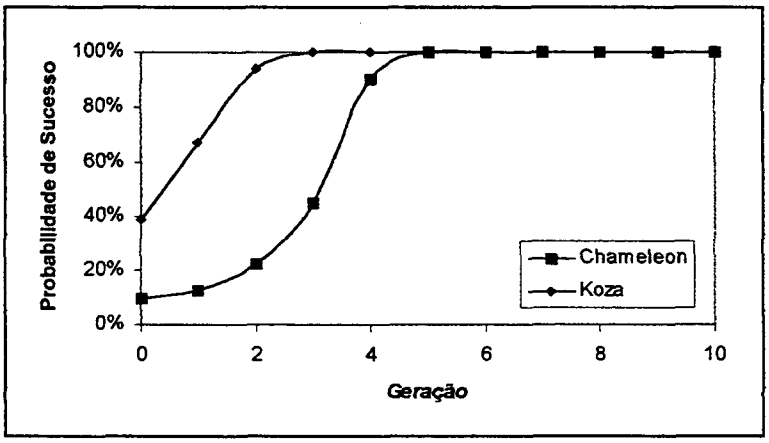

FIGURA 28:PROBABILIDADE DE SUCESSO PARA N = 3 COM ADF

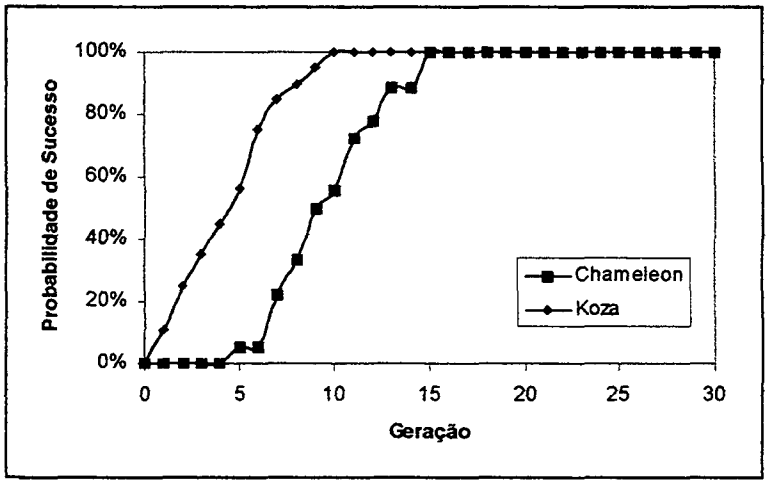

FIGURA 29:PROBABILIDADE DE SUCESSO PARA N = 4 COM ADF

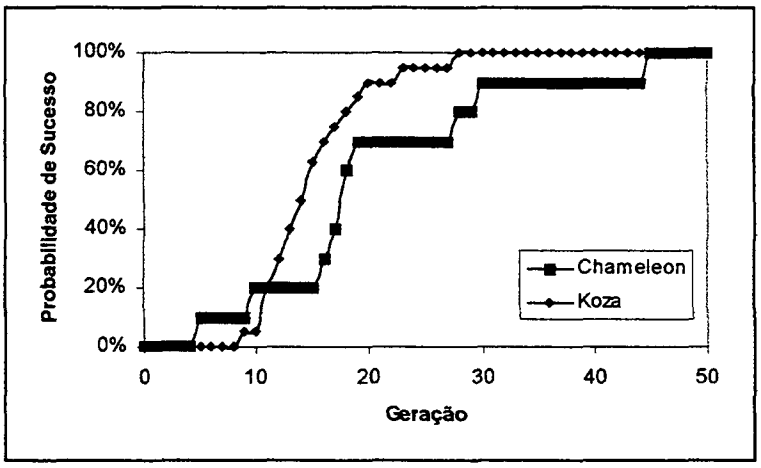

FIGURA 30:PROBABILIDADE DE SUCESSO PARA N = 5 COM ADF

Da mesma forma que o anterior, percebe-se que a convergência da ferramenta é mais lenta.

6.7 Eficiência de ADF em Programação Genética com Gramáticas

Para verificar a eficiência do uso de gramáticas em evolução de funções, um teste estatístico deve ser feito com base nos resultados obtidos. Para tal, foram estabelecidas as seguintes hipóteses:

- **Hipótese Nula (H_0):** o uso de ADF não melhora a convergência, isto é, a média da geração não é afetada pela uso de evolução de funções;

- **Hipótese Alternativa (H_1):** o uso de ADF melhora significativamente a convergência, isto é, a solução é encontrada mais cedo.

A Tabela 11 mostra os resultados obtidos na aplicação do teste *t* de Student. A conclusão foi baseada no uso de $\alpha = 5\%$ [Cohen 1995; Finn 1996].

Problema	Nível de Significância P	Conclusão
Paridade Par com $n = 3$	0.0357043	rejeitar H_0
Paridade Par com $n = 4$	$4.92430 \cdot 10^{-6}$	rejeitar H_0
Paridade Par com $n = 5$	$3.04624 \cdot 10^{-7}$	rejeitar H_0

TABELA 11: TESTE DE SIGNIFICÂNCIA

Pelos resultados obtidos, conclui-se que o ganho, para este caso, é estatisticamente significante, indicando que, mesmo com gramáticas, o uso de ADF facilita a convergência em problemas que apresentam escalabilidade. Isto é, quanto maior o espaço de busca, melhor é o ganho obtido, pelo que pode ser percebido pela redução do nível de significância *P* para aumentos no valor de *n*. Nas Figuras 31, 32 e 33 são apresentados os gráficos comparativos. Na Figura 33, o gráfico "Sem ADF" se refere somente as execuções que obtiveram sucesso até a 50ª. geração.

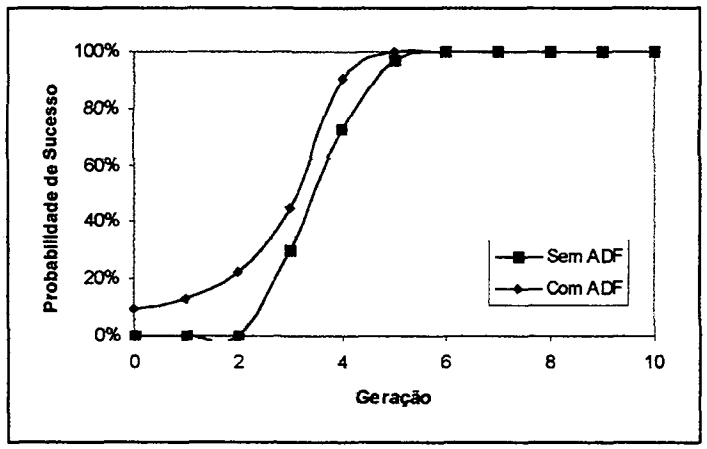

FIGURA 31: COMPARAÇÃO ENTRE USO OU NÃO DE ADF PARA N = 3

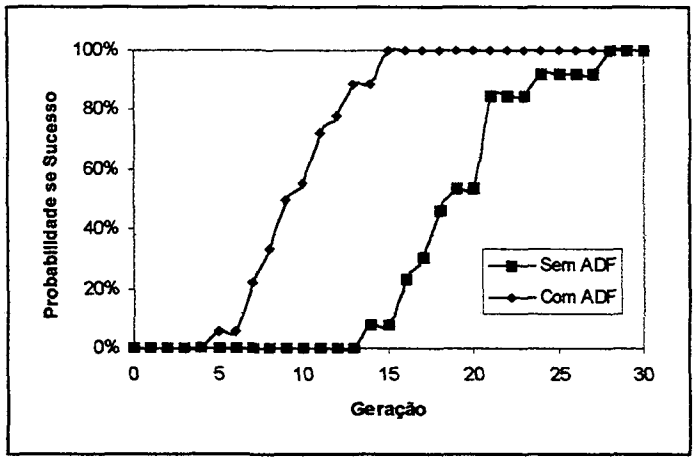

FIGURA 32: COMPARAÇÃO ENTRE USO OU NÃO DE ADF PARA N = 4

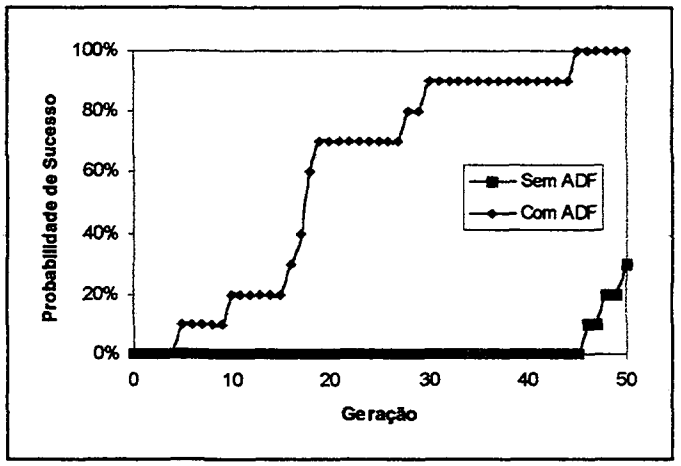

FIGURA 33: COMPARAÇÃO ENTRE USO OU NÃO DE ADF PARA N = 5

6.8 Discussão

Neste capítulo verificou-se inicialmente a viabilidade da ferramenta Chameleon através da comparação de resultados com os apresentados por John Koza [Koza 1994]. Isto foi feito mantendo a máxima semelhança entre as abordagens em termos de funcionamento, permitindo a comparação direta.

Pelos valores obtidos, percebemos que a ferramenta apresenta uma convergência levemente menor, o que não a inviabiliza pelo fato do processo de evolução ser estocástico, isto é, o comportamento evolutivo não é pré-determinado, podendo diferir levemente entre execuções distintas. Além disso, as diferentes formas de representação dos programas (árvores de sintaxe abstrata versus árvore de derivação) e a forma de atuação do operador de cruzamento (irrestrito em árvores de sintaxe abstrata) podem ter contribuído para as diferenças de convergência.

A eficiência do uso de evolução de funções foi claramente demonstrada, tanto através de teste de hipóteses como pelo gráficos. Percebeu-se que quanto maior n, melhor é a abordagem com ADF em comparação a abordagem sem ADF. Inclusive para $n = 5$ pode-se obter solução em todas as execuções com ADF, o mesmo não acontecendo sem o seu uso.

7. CONCLUSÕES

A adequação da Programação Genética a uma linguagem ou domínio específico, exige alterações no algoritmo. A abordagem baseada em Estruturas Sintáticas Restritas [Koza 1992] exige alterações no algoritmo para cada problema, não permitindo sua generalização. A Programação Genética Fortemente Tipada [Montana 1995] somente lida com a restrição de tipo de dado. Ambas não permitem o desenvolvimento de uma ferramenta realmente genérica.

Com o uso de gramáticas, torna-se fácil a adequação da Programação Genética a qualquer domínio ou linguagem de programação, simplesmente informando a gramática, sem necessidade de modificações no algoritmo. Desta forma, o uso de gramáticas permite o desenvolvimento de uma ferramenta de Programação Genética verdadeiramente genérica.

A viabilização de uma ferramenta genérica exige mudanças no algoritmo a fim de respeitar as regras impostas pela gramática. Neste trabalho foram apresentadas as modificações necessárias na representação dos programas, na criação da população inicial e na atuação dos operadores genéticos.

Uma ferramenta, Chameleon, foi apresentada e seus resultados são compatíveis com os apresentados em Programação Genética tradicional. Chameleon representa uma ferramenta útil de Programação Genética por permitir o uso de gramáticas e ADF, não existente em ferramentas de domínio público.

Exemplos de execução foram apresentados de forma a demonstrar a aplicabilidade de Chameleon em problemas sem restrição de formato (regressão simbólica) e com restrição de formato (Séries de Fourier). O problema do fatorial também foi apresentado, demonstrando o efetivo controle que Chameleon exerce sobre código anômalo (laço infinito ou estouro aritmético).

O uso de evoluções de funções também representa um papel fundamental na resolução de problemas mais difíceis, tais como o problema da paridade par demonstrado neste trabalho. Foi possível comprovar estatisticamente a eficiência do uso de funções automaticamente definidas (ADF) na Programação Genética com gramáticas.

Porém os resultados obtidos são levemente inferiores aos apresentados por John Koza. As diferentes estruturas de árvore dos programas e o mecanismo de geração da população inicial podem ser a causa destas diferenças.

7.1 Trabalhos Futuros

Como trabalhos futuros, propomos os seguintes:

- Adequação da ferramenta para outros métodos de evolução de funções (MA, ADM e ARL) e comparação destes métodos com os resultados obtidos com ADF;

- Verificar a eficiência da abordagem em domínios que envolvam movimentação de objetos tais como A Trilha da Formiga [Koza 1992] e o Cortador de Grama [Koza 1994];

- Implementar os outros métodos de geração da população inicial (*random-branch*, *uniform*, *PTC1* e *PTC2*) e verificar se há melhoria na convergência;

- Usar Chameleon para evoluir estruturas que possam ser descritas por uma gramática porém não sendo programas, tais como redes neurais;

A. GRAMÁTICAS

Neste anexo, aborda-se com detalhes as gramáticas, sua terminologia e estruturas sintáticas, bem como sua classificação. O material é um resumo do apresentado em [Aho 1995; Sethi 1996; Terry 1997] com várias complementações para efeito de completitude deste trabalho.

A.1 Introdução

Tanto as linguagens naturais (tais como o Português) quanto as linguagens de programação (tais como o C) são semelhantes em vários aspectos. Em ambas, as sentenças da linguagem são formadas por conjuntos de palavras e a construção destas sentenças seguem dois tipos de regras:

a) **Sintaxe:** descreve a forma das sentenças. Por exemplo, em Português, a sentença "eles podem pescar" está sintaticamente correta, enquanto que a sentença "podem pescar eles" está incorreta. A sintaxe de uma linguagem é expressa na forma de uma gramática.

b) **Semântica:** descreve o significado das sentenças sintaticamente corretas. A sentença "eles podem pescar" apresenta duas interpretações possíveis: (a) é permitido que eles pesquem ou (b) eles são capazes de pescar. A definição das regras semânticas são difíceis de se formular.

O objetivo deste anexo é definir como as gramáticas são formalizadas para fins de geração de programas de computador. Para tanto, alguns conceitos devem ser apresentados inicialmente.

A.2 Terminologia

- Um símbolo ou *token* é um elemento representado por um ou mais caracteres. Por exemplo: `+ , ; for`.

- Um alfabeto A é um conjunto não-vazio e finito de símbolos. Por exemplo, o alfabeto da linguagem C++ inclui o alfabeto `A = {+, -, *, /, if, switch}`

- Uma frase, palavra ou *string* de um alfabeto A é a seqüência $\sigma = a_1 a_2 ... a_n$ onde cada
  $a_i \in A$ para $i = \{ 1, 2, ..., n \}$

- É usual se definir a existência da *string* de tamanho zero chamada *string* nula denotado por ε. A *string* nulo tem como propriedade que a sua concatenação com qualquer *string*, seja pela esquerda ou direita, a *string* permanece inalterada.

$$a\varepsilon = \varepsilon a = a$$

- O conjunto de todas as *string* de tamanho n de um alfabeto A é denotada por A^n. O conjunto de todas as *strings* (incluindo a *string* nula) de um alfabeto A é chamado de *Kleene closure* ou simplesmente *closure*, e é denotado por A^*. O conjunto de todas as *strings* de tamanho maior que zero de um alfabeto A é chamado *positive closure* e é denotado por A^+. Portanto,

$$A^* = A^0 \cup A^1 \cup A^2 \cup ...$$

- A linguagem L de um alfabeto A é um subconjunto de A^*. Uma linguagem é simplesmente um conjunto de *strings*.

- A linguagem L pode ser definida simplesmente por uma lista de todas as *strings* que a compõem ou através de uma regra que as derive. Por exemplo:

$$L = \{ ([a^+)^n (b])^n \mid n > 0 \}$$

produz diversas *strings* tais como:

$[a+b]$

$[a+[a+b]b]$

$[a+[a+\ [a+b]b]b]$

A.3 Expressões Regulares

Algumas linguagens simples podem ser convenientemente especificadas usando a notação das expressões regulares. Uma expressão regular especifica a forma que uma palavra ou sentença pode assumir ao usar os símbolos do alfabeto A com alguns metasímbolos que representam:

- **Concatenação:** símbolos ou *strings* podem ser concatenados simplesmente escrevendo-os um ao lado do outro. Se houver necessidade de maior clareza, pode-se usar o metasímbolo · (ponto).

- **Alternância:** uma escolha entre dois símbolos a e b é indicado pela barra vertical |.

- **Repetição:** um símbolo a seguido do metasímbolo * indica que uma seqüência de zero ou mais ocorrências de a é permitida.

- **Agrupamento:** um grupo de símbolos podem ser envolvidos pelos metasímbolos (e)

Como um exemplo de uma expressão regular, considere:

$$1\,(1\,|\,0)^{*}\,0$$

Isto gera um conjunto de *strings*, cada uma tendo o símbolo 1 na frente, seguido por qualquer número de 0s ou 1s e terminado com o símbolo 0. O conjunto gerado por esta expressão regular é:

{ 10, 100, 110, 1000, ... }

Formalmente, as expressões regulares podem ser definidas como segue:

- Uma expressão regular denota um conjunto regular de *strings*.
- ϕ é uma expressão regular representando o conjunto vazio.
- ε é uma expressão regular representando o conjunto que contém apenas o *string* nulo.
- σ é uma expressão regular contendo apenas a *string* σ.
- Se A e B são expressões regulares, então (A) e $A \mid B$ e A^* são também expressões regulares.

Alguns pontos devem ser ressaltados:

- A aplicação dos metasímbolos seguem uma determinada precedência. Os parênteses tem precedência sobre a repetição (*) que, por sua vez, tem precedência sobre a concatenação (·). Por último, a concatenação tem precedência sobre a alternância (|). Por exemplo, as duas expressões regulares são equivalentes:

  eles | elas e el (e | a) s

- Se os metasímbolos são símbolos do alfabeto, a convenção é envolvê-los em aspas (") quando se referirem como símbolos. Por exemplo, comentários em Pascal podem ser descritos pela expressão regular:

  "(" "*" c^* "*" ")" sendo que $c \in A$

A.4 Estruturas Sintáticas

A sintaxe da maioria das linguagens de programação são muito mais complexas do que se pode expressar por expressões regulares. Existem duas formas possíveis de se expressar a sintaxe: a *abstrata* e a *concreta*.

A sintaxe abstrata identifica os componentes significativos de cada construção da linguagem diretamente. A expressão $a + b$ tem como componentes significativos: o operador + e as subexpressões a e b. A árvore de sintaxe abstrata correspondente está na Figura 34.

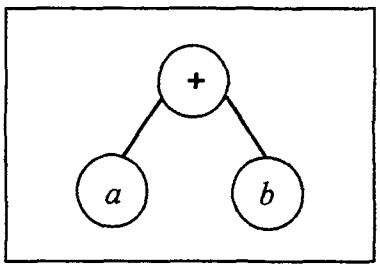

FIGURA 34: ÁRVORE DE SINTAXE ABSTRATA PARA A EXPRESSÃO $a + b$

Uma árvore consiste em um nó com k árvores como seus filhos, onde $k \geq 0$. Quando $k = 0$, a árvore consiste de apenas um nó, sem filhos. Um nó sem filhos é chamado de nó-folha. O nó-raiz ou, simplesmente, raiz de uma árvore é o nó que não apresenta "pai", isto é, não é filho de nenhum outro nó.

Se uma expressão é formada através da aplicação de um operador *op* para os operandos $E_1, E_2, \ldots, E_k$ para $k \geq 0$, então sua árvore tem um nó para *op* com k filhos, as árvores de cada subexpressão. A representação desta árvore está na Figura 35.

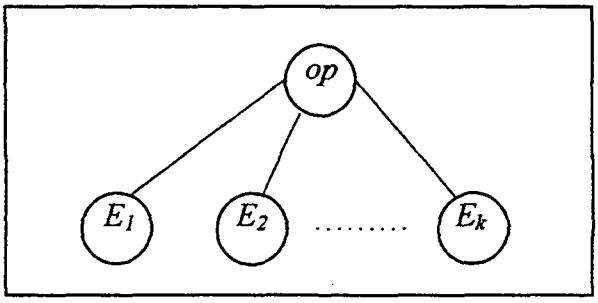

FIGURA 35: ÁRVORE GENÉRICA DE SINTAXE ABSTRATA

A sintaxe concreta descreve a representação da linguagem, incluindo detalhes léxicos tais como a colocação correta das palavras-chave e marcas de pontuação. A descrição de uma sintaxe concreta é feita através de gramáticas.

A.5 Gramáticas e Produções

Uma gramática é essencialmente um esquema de geração de infinitas *strings* (sentenças) de uma determinada linguagem. Portanto, uma gramática que define uma linguagem de programação pode ser considerada como a forma mais geral de se descrever qualquer programa nesta linguagem.

Basicamente, uma gramática é um conjunto de regras para descrever sentenças, isto é, escolher dos subconjuntos de A^*, um que nos interesse.

Formalmente, uma gramática G é a quádrupla $\{ N, \Sigma, S, P \}$ onde:

- N representa o conjunto finito de símbolos não-terminais;
- Σ representa o conjunto finito de símbolos terminais;
- S representa o símbolo inicial;
- P representa o conjunto finito de produções.

Uma sentença é formada somente por símbolos terminais escolhidos do conjunto Σ. Por outro lado, o conjunto N representa as classes sintáticas da gramática, isto é, os componentes usados para se descrever a construção das sentenças.

A união dos conjuntos N e Σ forma o vocabulário V de uma gramática ($V = N \cup T$) sendo que $N \cap \Sigma = \phi$ (são disjuntos).

Existem diversas formas de se especificar as produções. Essencialmente, uma produção especifica como uma *string* pode ser transformada em outra usando uma notação semelhante a $\gamma \rightarrow \delta$.

Suponha que σ e τ são duas *strings*, sendo que cada uma contem zero ou mais terminais ou não-terminais, isto é, $\sigma, \tau \in V = (N \cup \Sigma)^*$.

- Se for possível obter a *string* τ da *string* σ ($\sigma \rightarrow \tau$) aplicando uma das produções da gramática *G*, então afirma-se que σ diretamente produz τ e expressa-se como $\sigma \Rightarrow \tau$.

  Por exemplo, se $\sigma = \alpha \delta \beta$ e $\tau = \alpha \gamma \beta$, se existe a produção $\delta \rightarrow \gamma$ em *G*, então $\sigma \Rightarrow \tau$.

- Se for possível obter a *string* τ da *string* σ aplicando-se *n* produções de *G*, com $n \geq 1$, então diz-se que σ produz τ de forma não-trivial e expressa-se como $\sigma \Rightarrow^+ \tau$

  Por exemplo, se existe uma seqüência $\alpha_0, \alpha_1, \alpha_2, \ldots \alpha_k$ com $k \geq 1$ de tal forma que $\sigma \rightarrow \alpha_0$, $\alpha_{j-1} \rightarrow \alpha_j$ ($1 \leq j \geq k$) e $\alpha_k \rightarrow \tau$, então $\sigma \Rightarrow^+ \tau$

- Se for possível produzir a *string* τ da *string* σ aplicando-se *n* produções de *G*, com $n \geq 0$ (incluindo a anterior e mais o caso $\sigma \Rightarrow \tau$), então afirma-se que σ produz τ e expressa-se por $\sigma \Rightarrow^* \tau$

- Uma forma sentencial é qualquer *string* que possa ser produzida, isto é, qualquer *string* σ tal que $S \Rightarrow^* \sigma$

- Uma gramática é dita recursiva se permitir derivações da forma $A \Rightarrow^+ \varpi_1 A \varpi_2$ sendo que $A \in N$ e $\varpi_1, \varpi_2 \in V^*$. Mais especificamente, é chamada recursiva à esquerda se $A \Rightarrow^+ A\varpi$, e recursiva à direita se $A \Rightarrow^+ \varpi A$.

- Formalmente, define-se a linguagem *L(G)* produzida pela gramática *G* pela relação:

$$L(G) = \{ w \mid w \in \Sigma^* ;\ S \Rightarrow^* w \}$$

A.6 A Hierarquia de Chomsky

Basicamente, uma gramática é caracterizada pela forma de suas produções. Dependendo do formato das produções, as gramáticas são classificadas em quatro classes [Chomsky 1959]:

Gramáticas Irrestritas (Tipo 0)

Uma gramática é dita irrestrita se não impor nenhuma restrição na forma de qualquer produção, tendo a forma geral

$$\gamma \rightarrow \delta \quad \text{sendo} \quad \gamma \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \quad \delta \in (N \cup \Sigma)^*$$

Sendo que a única restrição é que deva haver ao menos um não-terminal no lado esquerdo de cada produção. Além disso, para se qualificar como um gramática do tipo 0, é necessária a presença de ao menos uma produção $\gamma \rightarrow \delta$ onde $|\gamma| > |\delta|$, sendo que $|\gamma|$ denota o tamanho (número de símbolos) de γ. Tal produção pode ser usada para "remover" símbolos, por exemplo $aAB \rightarrow aB$ remove A do contexto aAB. Este tipo de gramática não tem aplicação em linguagens de programação.

Gramáticas Sensíveis a Contexto (Tipo 1)

Se for imposta a restrição de que o tamanho da *string* a esquerda de qualquer produção seja menor ou igual ao tamanho da *string* a direita, a gramática denomina-se *sensível a contexto*. Sua forma geral é:

$$\gamma \rightarrow \delta \quad \text{com} \, |\gamma| \leq |\delta|, \text{ sendo } \quad \gamma \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \quad \delta \in (N \cup \Sigma)^+$$

Intuitivamente é possível perceber que a substituição de um não-terminal pode estar condicionada a presença de outros símbolos ao seu redor.

Gramáticas Livre de Contexto (Tipo 2)

Se restringir mais ainda as produções de forma a conterem somente não-terminais no seu lado esquerdo e uma seqüência não-vazia de terminais e não-terminais a sua direita, obtem-se uma gramática *livre de contexto*. Sua forma geral é:

$$\gamma \rightarrow \delta \quad \text{sendo } \quad \gamma \in N \, , \quad \delta \in (N \cup \Sigma)^+$$

As gramáticas livre de contexto são as mais comuns devido ao fato de serem simples e facilmente aplicáveis a vários problemas em programação. A grande maioria das linguagens de programação podem ser definidas usando este tipo de gramática.

Gramáticas Regulares (Tipo 3)

As gramáticas regulares apresentam duas formas: regulares à direita e regulares à esquerda. Uma gramática regular à direita apresenta o lado direito de qualquer produção consistindo de zero ou mais símbolos terminais, opcionalmente seguido por um não-terminal. Seu lado esquerdo é sempre um não-terminal. Sua forma geral é:

$$A \rightarrow a \quad \text{ou} \quad A \rightarrow aB \quad \text{sendo} \quad a \in \Sigma \; ; \; A, B \in N$$

No caso das gramáticas regulares à esquerda, o lado direito de qualquer produção consiste de zero ou mais símbolos terminais, opcionalmente precedido por um não-terminal. Seu lado esquerdo é sempre um não-terminal. Sua forma geral é:

$$A \rightarrow a \quad \text{ou} \quad A \rightarrow Ba \quad \text{sendo} \quad a \in \Sigma \; ; \; A, B \in N$$

As gramáticas regulares são muito restritivas, servindo unicamente para descrever características específicas de linguagem de programação tais como as definições de identificadores ou números. Tais gramáticas apresentam a propriedade de que suas sentenças podem ser desmembradas por autômatos de estados finitos e podem, alternativamente, serem descritas por expressões regulares.

A Relação entre Tipo de Gramática e Tipo de Linguagem

É possível perceber que as gramáticas regulares (tipo 3) são um subconjunto das gramáticas livre de contexto (tipo 2) que, por sua vez, são um subconjunto das gramáticas sensíveis ao contexto (tipo 1). Da mesma forma, as gramáticas sensíveis ao contexto (tipo 1) são um subconjunto das gramáticas irrestritas (tipo 0). A hierarquia das gramáticas está na Figura 36.

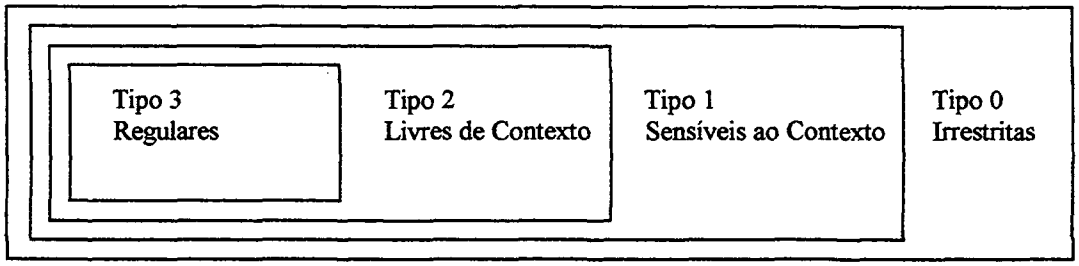

FIGURA 36: HIERARQUIA DAS GRAMÁTICAS

Uma linguagem *L(G)* é do tipo *k* se pode ser gerada por uma gramática do tipo *k*. Então, por exemplo, uma linguagem é dita livre de contexto se uma gramática livre de contexto possa ser usada para defini-la. É importante frisar que o simples fato de não existir uma gramática livre de contexto para uma determinada linguagem, não implica necessariamente que a linguagem não é livre de contexto. Pode eventualmente existir uma gramática livre de contexto, ainda a ser descoberta, que possa descrevê-la.

Atualmente, as linguagens de programação modernas são livres de contexto, com algumas inevitáveis características sensíveis a contexto. Estas características são manipuladas por regras separadas e através do uso das chamadas gramáticas atributivas (*attribute grammars*). Estas características incluem a declaração de uma variável deve preceder seu uso, o número de parâmetros formais e atuais em um procedimento devem ser iguais dentre outras.

Notação BNF

Existem diversas formas de se representar as regras de produção de uma gramática, porém uma notação largamente usada, principalmente em gramáticas livre de contexto, é a notação *Bakus-Naur Form* (BNF).

Em BNF, um não-terminal é identificado por um nome envolvido pelos símbolos < e >. Qualquer nome não envolvido por < e > é considerado um terminal. As produções tem a forma:

$$lado\_esquerdo \rightarrow derivação$$

O símbolo → é interpretado como "produz"[22]. Em tais produções, tanto *lado_esquerdo* quanto *derivação* consistem em uma seqüência de um ou mais terminais e não-terminais. De fato, usando a notação anterior,

$$lado\_esquerdo \in (N \cup \Sigma)^+ \quad \text{e} \quad derivação \in (N \cup \Sigma)^*$$

[22] Alguns textos usam o símbolo ::= no lugar de →.

Sendo que necessariamente *lado_esquerdo* $\cap$ $N \neq \Phi$, isto é, o *lado_esquerdo* deve conter ao menos um não-terminal. De forma semelhante às expressões regulares, as produções da forma $x \rightarrow y$ e $x \rightarrow z$ podem ser expressas com o uso do símbolo | (barra vertical), tal como $x \rightarrow y \mid z$.

Como exemplo, considere a gramática $G = \{ N, \Sigma, S, P \}$ onde

Σ = { x, +, -, *, / }
N = { <exp>, <op>, <var> }
S = <exp>
P = { <exp> → <var> |
(<exp> <op> <exp>)
<op> → + | - | * | /
<var> → x }

Para se proceder a geração de uma sentença de G, inicia-se por S e aplica-se sucessivamente as produções até obter somente terminais. Por exemplo, a sentença (x-x) pode ser gerada pelos seguintes passos:

- Primeiro, escreve-se o símbolo inicial (S)
  <exp>
- Substitui-se <exp> pela segunda derivação:
  <exp> ⇒ (<exp> <op> <exp>)
- Substitui-se o primeiro <exp> pela sua primeira derivação:
  (<exp> <op> <exp>) ⇒ (<var> <op> <exp>)
- O não-terminal <var> é então substituído pela única derivação possível:
  (<var> <op> <exp>) ⇒ (x <op> <exp>)
- Continua-se a resolver substituindo o não-terminal <op> pela sua terceira derivação:
  (x <op> <exp>) ⇒ (x * <exp>)
- O último não-terminal é substituído pela sua primeira derivação:
  (x * <exp>) ⇒ (x * <var>)

- Finalmente, substitui-se o não-terminal `<var>` por sua única derivação:

  `(x * <var>)` ⇒ `(x * x)`

A derivação da sentença `(x*x)` pode ser vista através de sua árvore de derivação[23] como mostrada na Figura 37.

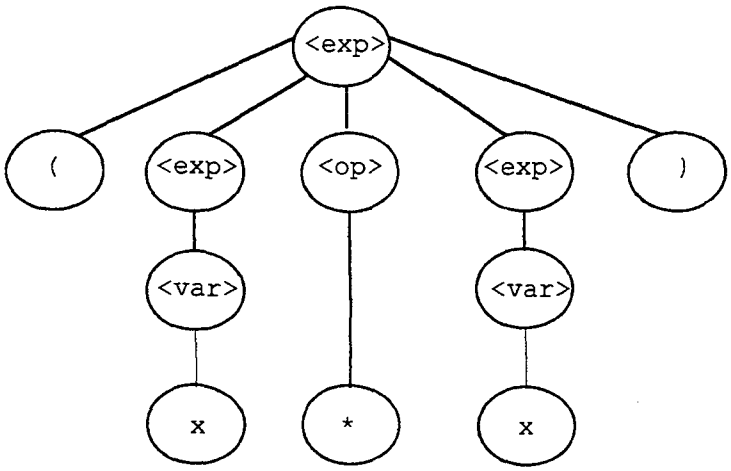

FIGURA 37: ÁRVORE DE DERIVAÇÃO PARA (X*X)

Portanto, se for necessário produzir uma sentença de G, inicia-se por S e sucessivamente substitui cada ocorrência de não-terminal até que não seja mais possível aplicar nenhuma produção. Tradicionalmente, substitui-se inicialmente os não-terminais mais à esquerda até se compor a sentença (*canonical derivation*), conforme exemplo apresentado.

[23] Também conhecida como Árvore de Sintaxe Concreta.

B. FUNÇÕES IMPLEMENTADAS EM CHAMELEON

Grande parte das as funções usadas por John Koza em seus dois livros [Koza 1992; Koza 1994] estão implementadas no interpretador de Chameleon, com exceção daquelas que lidam com restrição de formato e com efeitos colaterias. Na Figura 38 tem-se a listagem do arquivo `Biblio.h` com os protótipos das funções.

```
/*************************************************************
 *  Chameleon - Grammar-Guided Genetic Programming System
 *  Version..: 2.0
 *  Header...: Biblio
 *  Author...: Ernesto Rodrigues
 *  Updated..: November, 2001
 *************************************************************/

#include <math.h>
#include <float.h>
#include <stdarg.h>

// symbolic regression
double Sum(double value, ...);
double Sub(double value, ...);
double Mul(double value, ...);
double Div(double value, ...);

// boolean
double Not(double value, ...);
double And(double value, ...);
double Nand(double value, ...);
double Or(double value, ...);
double Nor(double value, ...);

// conditionals
double If(double value, ...);
double Gt(double value, ...);
double Lt(double value, ...);
double Eq(double value, ...);

// trigonometric
double Sin(double value, ...);
double Cos(double value, ...);
double xsin(double value, ...);
double xcos(double value, ...);

// others
double Sqrt(double value, ...);
double Log(double value, ...);
double W(double value, ...);
double P(double value, ...);
```

FIGURA 38: PROTÓTIPOS DAS FUNÇÕES PRESENTES EM CHAMELEON

REFERÊNCIAS BIBLIOGRÁFICAS

[Aho 1995] AHO, A. V.; ULLMAN, J. D. & SETHI, R.. Compiladores: princípios, técnicas e ferramentas . ISBN 8521610572. LTC, 1995.

[Finn 1996] FINN, J. D. & ANDERSON, T. W. The new statistical analysis of data. ISBN 0387946195. Springer-Verlag, 1996.

[Andre 1994] ANDRE, D. Evolution of map making: learning, planning, and memory using genetic programming. Proceedings of the First IEEE Conference on Evolutionary Computation. Vol I. pp. 250-255. IEEE Press, 1994.

[Angeline 1993] ANGELINE, P. J. & POLLACK, J. Evolutionary module acquisition. Proceedings of the 2nd Annual Conference on Evolutionary Programming, pp. 154-163, 1993.

[Angeline 1994] ANGELINE, P. J. Genetic programming and emergent intelligence. Advances in Genetic Programming, ISBN 0262111888. pp 75-98. MIT Press, 1994.

[Banzhaf 1998] BANZHAF, W; NORDIN, P.; KELLER, R. E. & FRANCONE, F. D. Genetic Programming: an introduction. ISBN 155860510X. Morgan Kaufmann, 1998.

[Baker 1989] BAKER, J. E. An analysis of the the effects of selection in genetic algorithms PhD thesis, Graduate Schol of Vanderbilt University. Nashiville, Tennessee, 1989.

[Barreto 1997] BARRETO, J. Inteligência Artificial. No Limiar do Século XXI. ISBN 859003822X. ρρρ Edições, 1997.

[Blickle 1995] BLICKLE, T. & THIELE, L. A comparision of selection schemes used in genetic algorithms. TIK-Report nr 11 version 2, Computer Engineering and Communication Networks Lab. Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, December, 1995.

[Bohm 1996] BOHM, W. & GEYER-SCHULZ, A. Exact unifrom initialization for genetic programming. Foundations of Genetic Algorithms IV (FOGA 4). ISBN 155860460X. pp 379-407. Morgan Kaufmann, 1996.

[Brameier 1998] BRAMEIER, M.; KANTSCHIK, W.; DITTRICH, P. & BANZHAF, W. SYSGP: A C++ library of different GP variants. Series Computational Intelligence, Internal Report of SFB 531, University of Dortmund, ISSN 1433-3325, Dortmund, 1998.

[Bruce 1995] BRUCE, W. S. The application of genetic programming to the automatic generation of object-oriented programs. PhD thesis. School of Computer and Information Sciences, New Southeastern University, 1995

[Chellapilla 1997] CHELLAPILLA, K. Evolving computer programs without sub-tree crossover IEEE Transactions on Evolutionary Computation. Vol. 1 Issue 3. pp 209-216. IEEE Press, September, 1997.

[Cohen 1995] COHEN, P. R. Empirical methods for artificial intelligence. ISBN 0262032252. MIT Press, 1995.

[Daida 1999] DAIDA, J. M. Challenges with verification, repeatability and meaningful comparisions in genetic programming. Proceedings of the 4th Annual Conference in Genetic Programming (GECCO'99). ISBN 1558606114. pp. 1069-1076. Morgan Kaufmann, 1999.

[Dessi 1999] DESSI, A.; GIANI, A. & STARITA, A. An analysis of automatic subroutine discovery in genetic programming. Proceedings of the 4th Annual Conference in Genetic Programming (GECCO'99). ISBN 1558606114. pp. 996-1001. Morgan Kaufmann, 1999.

[Franco 2001] FRANCO, L. & CANNAS, S. A. Generalization properties of modular networks: implementing the parity function. IEEE Transactions on Neural Networks, Vol 12, Issue 6, pp. 1306-1313. IEEE Press. November, 2001.

[Fraser 1994] FRASER, A. P. A genetic programming in C++. Technical Report 040. Cybernetics Research Intitute, University of Salford.

[Gathercole 1997] GATHERCOLE, C. & ROSS, P. Tackling the boolean even-*n*-parity problem with genetic programming and limited-error fitness. Proceedings of the Second Annual Conference in Genetic Programming (GECCO'97) ISBN 1558604838. pp. 119-127. Morgan Kaufmann, 1997.

[Gathercole 1998] GATHERCOLE, C. An investigation of supervised learning in genetic programming. PhD thesis. University of Edinburgh, 1998.

[Goldberg 1991] GOLDBERG, D. E. & DEB, K. A comparative analysis of selection schemes used in genetic algorithms. Foundations of Genetic Algorithms (FOGA). ISBN 1558601708. pp 69-93. Morgan Kaufmann, 1991.

[Gritz 1993] GRITZ, L. A C++ implementation of genetic programming. Department of Electrical Engeneering and Computer Science. The George Washington University, Washington, DC.

[Gruau 1995] GRUAU, F. & WHITLEY, D. A programming language for artificial development. Proceedings of the Fourth Annual Conference on Evolutionary Programming, San Diego, CA. pp 415-434. MIT Press, 1995.

[Gruau 1996] GRUAU, F. On syntactic constraints with genetic programming. Advances in Genetic Programming II. ISBN 0262011581. pp. 377-394. MIT Press, 1996.

[Holland 1975] HOLLAND, J. H. Adaptation in natural and artificial systems. The University of Michigan Press, Ann Arbor, MI, 1975.

[Horner 1996] HORNER, H. A C++ class library for genetic programming. Technical Report. The Vienna University of Economics, Vienna, Austria, 1996.

[Kinnear Jr 1994] KINNEAR, K. E. Alternatives in automatic function definition: a comparision of performance. Advances in Genetic Programming, ISBN 0262111888. pp 119-141. MIT Press, 1994.

[Koza 1989] KOZA, J. R. Hierarquical genetic algorithms operating on populations of computer programs. Proceedings of the 11th International Joint Conference on Artificial Intelligent (IJCAI-89). Detroit, MI. pp 768-774. Morgan Kaufmann, 1989.

[Koza 1992] KOZA, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. ISBN 0262111705. MIT Press, 1992.

[Koza 1994] KOZA, J. R. Genetic Programming II: Automatic Discovery of Reusable Programs. ISBN 0262111896. MIT Press, 1994

[Kramer 2000] KRAMER, M. D. & ZHANG, D. GAPS: a genetic programming system. The 24th Annual International Computer Software and Applications Conference 2000 (COMPSAC 2000) ISBN 0769507921. pp 614–619. IEEE Press, October, 2000.

[Lee 1997] LEE, R. C. & TEPFENHART, M. UML and C++ - a practical guide to object-oriented development. ISBN 0130290408. 2nd Edition. Prentice Hall, 1997

[Langdon 1996] LANGDON, W. B. Genetic Programming and data structures PhD thesis. Department of Computer Science. University of London, 1996.

[Luke 2000] LUKE, S. Two fast tree-creation algorithms for genetic programming. IEEE Transactions in Evolutionary Computation, Vol 4 Issue 3 pp. 274-283 IEEE Press. September, 2000.

[Luke 2001] LUKE, S. & PAINAIT, L. A survey and comparison of tree generation algorithms. Proceedings of the 6th Annual Conference in Genetic Programming (GECCO 2001). ISBN 1558607749. Springer-Verlag, 2001.

[Montana 1994] MONTANA, D. J. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Cambridge, MA, 1994.

[Montana 1995] MONTANA, D. J. Strongly typed genetic programming. Evolutionary Computation. Vol 3 Issue 2. ISBN 10636560. pp 199-230. MIT Press, 1995.

[Muhlenbein 1993] MUHLENBEIN, H. & SCHIERKAMP-VOOSEN, D. Predictive models for the breeder genetic algorithms. Evolutionary Computation Vol 1 Issue 1, ISBN 10636560. pp 25-49. MIT Press, 1993.

[Musser 2001] MUSSER, D. R.; DERGE, G. J. & SAINI, A. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. ISBN 0201379236. Addison-Wesley, 2001.

[O'Neil 2000] O'NEIL, M. & RYAN, C. Grammar based function definition in Grammatical Evolution. Proceedings of the 5th Annual Conference in Genetic Programming. (GECCO 2000) ISBN 1558607080. pp. 485-490. MIT Press, 2000.

[O'Reilly 1995] O'REILLY, U. An analysis of Genetic Programming. PhD thesis. Ottawa-Carleton Institute for Computer Science, Carleton University, 1995.

[O'Reilly 1996] O'REILLY, U. Investigating the generality of automatically defined functions. Proceedings of the 1st. Annual Conference in Genetic Programming. ISBN 0262611279. MIT Press, 1996

[Poli 1999] POLI, R.; PAGE, J. & LANGDON, W. B. Smooth uniform crossover, sub-machine code GP and demes: a recipe for solving high-order boolean parity problem. Proceedings of the Genetic and Evolutionary Computation Conference, Vol 2, pp. 1162-1169. Morgan Kaufmann, 1999

[Ratle 2000] RATLE, A. & SEBAG, M. Genetic programming and domain knowledge: beyond the limitations of grammar-guided machine discovery. In Proceedings of the Sixth Conference on Parallel Problems Solving from Nature, LNCS, pp. 211-220. Springer-Verlag, 2000

[Rodrigues 2001] RODRIGUES, E.; POZO, A.; VERGILIO, S. & MUSICANTE, M. Chameleon: uma ferramenta de indução de programas. SCCC 2001 – II Workshop en Inteligencia Artificial. ISBN 0769513964. Punta Arenas, Chile, November, 2001. IEEE Computer Society, 2001.

[Rosca 1996] ROSCA, J. P. & BALLARD, D. H. Discovery of subroutines in genetic programming. Advances in Genetic Programming II. ISBN 0262011581. pp 177-201. MIT Press, 1996.

[Rosca 1997] ROSCA, J. P. Hierarquical learning with procedural abstraction mechanisms. PhD thesis. University of Rochester, Rochester, NY, 1997.

[Rumelhart 1986] RUMELHART, D.; HILTON, G. & WILLIAMS, R. Learning internal representations by error propagation. Parallel Distributed Processing: Explorations in the Micro-structures of Cognition. MIT Press, 1986.

[Ryan 1998] RYAN, C.; COLLINS, J. J. & O'NEIL, M. Grammatical evolution: evolving programs for an arbitrary language. Proceedings of the First Workshop on Genetic Programming, 1998 Vol. LNCS 1391, pp. 83-96. Springer-Verlag, 1998.

[Sethi 1996] SETHI, R. Programming languages concepts and constructs. ISBN 0201590654. Addison-Wesley, 1996.

[Spector 1995] SPECTOR, L. Evolving control structures with automatically defined macros. Working Notes of the AAAI Fall Symposium on Genetic Programming 1995. The American Association for Artificial Intelligence. pp. 99-105, 1995.

[Spector 1996] SPECTOR, L. Simultaneous evolution of programs and their control structures. Advances in Genetic Programming II.. ISBN 0262011581. pp 137-154. MIT Press, 1996.

[Terada 1991] TERADA, R. Desenvolvimento de algoritmos e estruturas de dados. ISBN 0074609602. McGraw-Hill, Makron, 1991.

[Terry 1997] TERRY, P. D. Compilers and Compiler Generators, an introduction with C++. ISBN 1850322988. International Thomson Computer Press, 1997.

[Whigham 1995] WHIGHAM, P. A. Grammatically based genetic programming. Proceedings of ML'95 Workshop on Genetic Programming. From Theory to Real-World Applications. pp 33-41. Morgan Kauffmann, 1995.

[Whigham 1996] WHIGHAM, P. A. Grammatical Bias for Evolutionary Learning. Ph.D. thesis. School of Computer Science. University of New South Wales, Australian Defense Force Academy, 1996.

[Willis 1997] WILLIS, M. J.; HIDEN, H. G.; MARENBACH , P.; MCKAY, B. & MONTAGE, G. A. Genetic programming: an introduction and survey of applications. The Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications 1997 (GALESIA 97). pp 314-319. IEEE Press, 1997

[Wong 1996] WONG, M. L. & LEUNG, K. S. Evolving recursive functions for the even-parity problem using genetic programming Advances in Genetic Programming II. ISBN 0262011581. pp. 222-240. MIT Press, 1996.

[Yu 1999] YU, T. Structure Abstraction and Genetic Programming. Proceedings of the 1999 Congress on Evolutionary Computation (CEC99). ISBN 0780355369 pp. 652-659. IEEE Press, 1999.

[Zongker 1996] ZONKGER, D.; PUNCH, B. & RAND, B. Lil-gp 1.01 User's manual Genetic Algorithms Research and Applications Group (GARAGe), Department of Computer Science, Michigan State University, 1996.