UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CUROS DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

O USO DE PROGRAMAÇÃO GENÉTICA NA TOMADA DE DECISÃO EM JOGOS

ANDRÉ LUIS GALASTRI

BLUMENAU
2003

2003/2-11

ANDRÉ LUIS GALASTRI

O USO DE PROGRAMAÇÃO GENÉTICA NA TOMADA DE DECISÃO EM JOGOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos - Orientador

BLUMENAU
2003

2003/2-11

O USO DE PROGRAMAÇÃO GENÉTICA NA TOMADA DE DECISÃO EM JOGOS

Por

ANDRÉ LUIS GALASTRI

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos – Orientador, FURB
Membro:	Prof. Jomi Fred Hubner, FURB
Membro:	Prof. Roberto Heinzle, FURB

Blumenau, dia 18 de Novembro de 2003

Dedico este trabalho a todos aqueles que, direta ou indiretamente, contribuíram para a sua realização.

Computadores estão para as Ciências da
Computação da mesma maneira que
telescópios estão para a Astronomia.

E. W. Djikstra

AGRADECIMENTOS

Agradeço a meus professores, os da pré-escola por terem tido paciência, os do primeiro grau por terem se preocupado, os do segundo grau por ensinarem que é importante me preocupar e os da faculdade por ensinarem que não vale a pena se preocupar.

A todos os outros “professores” que, intencionalmente ou não, me ensinaram muitas lições.

RESUMO

O presente trabalho aborda os conceitos de programação genética e descreve a implementação de um protótipo de *Software* que demonstra estes conceitos através da construção de uma aplicação na área de programação genética na tomada de decisão em jogos. O foco da aplicação foi ”evoluir” um programa controlador de tanques para o simulador Robocode.

Palavras chaves: Programação genética; Robocode; Java.

ABSTRACT

This work refers the main concepts of the genetic programming and describes a software prototipe that validates those concepts by building an aplication in game decision making. The main goal of the aplication was the generation of tank controler program to the Robocode simulator.

Key-Words: Gennetic programming; Robocode; Java.

LISTA DE ILUSTRAÇÕES

FIGURA 2.1 - Fluxograma da programação genética 15
FIGURA 2.2 – Um trecho de código em Java e sua árvore 16
FIGURA 2.3 - Exemplo de programas 18
FIGURA 2.4 – Equação do volume de uma esfera 19
FIGURA 2.5 – Formula do *adjusted fitness* 21
FIGURA 2.6 – Formula da *normalized fitness* 22
FIGURA 2.7 – “Roleta” de uma seleção proporcional a *fitness* 23
FIGURA 2.8 – Árvores e nós escolhidos para *crossover* 24
FIGURA 2.9 – Árvores após o *crossover* 25
FIGURA 2.10 - *Crossover* entre terminal e sub-árvore 26
FIGURA 2.11 - *Crossover* entre terminal e terminal 26
FIGURA 2.12 - *Crossover* entre raiz e terminal 27
FIGURA 3.1 – Anatomia de uma tanque no Robocode. Onde *Gun* é o canhão, *Vehicle* é o corpo e *Radar* é o próprio radar 29
FIGURA 3.2 – Dois tanques “duelando” no simulador Robocode. 32
FIGURA 3.3 – Uma batalha envolvendo vários tanques. 33
FIGURA 3.4 – Código fonte do tanque “Fire” 36
FIGURA 3.5 – Código fonte do tanque “Crazy” 37
FIGURA 4.1 – Diagrama de classes 40
FIGURA 4.2 – Fragmento do diagrama de classes UML centrado na classe Engine 41
FIGURA 4.3 – Fragmento do diagrama de classes UML centrado na classe Population 42
FIGURA 4.4 – Fragmento do diagrama de classes UML centrado na classe ClassGenerator 43
FIGURA 4.5 – Funcionamento do protótipo 44
FIGURA 4.6 – Código fonte da *interface* Node 46
FIGURA 4.7 – Código fonte da classe Evaluator 50
FIGURA 4.8 – Formato do arquivo XML 51
FIGURA 5.1 – Primeira parte do arquivo XML utilizado com o Robocode. 55
FIGURA 5.2 – Segunda parte do arquivo XML utilizado com o Robocode. 56
FIGURA 5.3 – Fragmento do código fonte da classe avaliadora dos tanques. 57
FIGURA 5.4 – Divisão aproximada do tempo de cada geração. 59
FIGURA 5.5 – Histórico de evolução média da *adjusted fitness* 60
FIGURA 5.6 – Indivíduo da população inicial 60
FIGURA 5.7 – Melhor indivíduo encontrado. 61

LISTA DE TABELAS

TABELA 2.1 –Exemplos de *raw fitness.* 20

SUMÁRIO

1 INTRODUÇÃO ... 11

1.1 OBJETIVOS DO TRABALHO ... 12

1.2 ESTRUTURA DO TRABALHO ... 12

2 PROGRAMAÇÃO GENÉTICA ... 13

2.1.1 CONCEITO ... 13

2.1.2 FUNÇÃO E OBJETIVO ... 13

2.1.3 FUNCIONAMENTO ... 14

2.1.4 LINGUAGEM DE PROGRAMAÇÃO, FUNÇÕES E TERMINAIS ... 15

2.1.4.1 PROPRIEDADE CLOSURE ... 17

2.1.4.2 PROPRIEDADE SUFICIENCIA ... 18

2.1.4.3 POPULAÇÃO INICIAL ... 19

2.1.5 AVALIAÇÃO DOS PROGRAMAS - FITNESS ... 19

2.1.5.1 RAW FITNESS ... 20

2.1.5.2 STANDARIZED FITNESS ... 20

2.1.5.3 ADJUSTED FITNESS ... 21

2.1.5.4 NORMALIZED FITNESS ... 21

2.1.6 OPERAÇOES GENETICAS ... 22

2.1.6.1 REPRODUÇÃO ... 22

2.1.6.2 CROSSOVER ... 23

2.1.7 CONDIÇÃO DE TÉRMINO ... 28

2.1.8 RESULTADO DA PROGRAMAÇÃO GENÉTICA ... 28

2.1.9 TRABALHOS CORRELATOS ... 28

3 ROBOCODE ... 29

3.1.1 FUNCIONAMENTO ... 29

3.1.2 ANATOMIA DO TANQUE ... 29

3.1.3 REGRAS DO JOGO ... 31

3.1.4 CAMPO DE BATALHA ... 31

3.1.5 BATALHA E PONTUAÇÃO ... 32

3.1.6 ESTRUTURA DO PROGRAMA CONTROLADOR ... 34

3.1.7 EXEMPLO DE ROBO ... 35

4 ESPECIFICAÇÃO DO SISTEMA ... 38

4.1 REQUISITOS DO SISTEMA ... 38

4.2 ESPECIFICAÇÃO DA FERRAMENTA JNETIC 38
4.3 IMPLEMENTAÇÃO 44
4.3.1 FUNÇOES E TERMINAIS 45
4.3.2 POPULAÇÃO INICIAL 47
4.3.3 REPRODUÇÃO 48
4.3.4 CROSSOVER 48
4.3.5 DESCRIÇÃO DO ARQUIVO FONTE JAVA 49
4.3.6 AVALIAÇÃO DA FITNESS 49
4.3.7 PARAMETROS GENÉTICOS 50
4.3.8 UTILIZAÇÃO 54
5 ESTUDO DE CASO 55
5.1 FUNÇÕES, TERMINAIS E EVENTOS UTILIZADOS 55
5.2 CLASSE AVALIADORA 57
5.3 CONDIÇÃO DE TÉRMINO 58
5.4 RESULTADOS 59
6 CONCLUSÕES 62
REFERÊNCIAS BIBLIOGRÁFICAS 64

1 INTRODUÇÃO

Programação genética é um mecanismo de busca onde o universo é o conjunto de todos os programas de computador e o resultado final da busca é um único programa. Pode-se procurar, por exemplo, por um programa que tome como entrada dois números e retorne o resultado da soma de ambos, ou um programa que receba uma seqüência de números e retorne a mesma seqüência ordenada. A procura pelo programa alvo é realizada através de sucessivas seleções de indivíduos que representam, ou geram, os resultados mais próximos do esperado. Segundo Koza (1992), o universo dos programas de computadores possui uma quantidade praticamente ilimitada de elementos, e a programação genética é uma forma de encontrar elementos neste universo.

Os programas encontrados através da programação genética são "empíricos", ou seja, os resultados que geram não são o fruto de uma teoria ou estudo que foi transformado em algoritmo, mas sim um simples programa que retornou resultados corretos para os valores com os quais foi testado (Koza, 1992). De acordo com Koza (1992) é exatamente a natureza "inexata" de tais programas que fizeram com que o seu desenvolvimento e utilização fossem, de certa forma, relegados a um segundo plano em diversas áreas de Inteligência Artificial (IA). A tomada de decisão em jogos é uma destas áreas, embora seja uma aplicação clássica de IA, é, paradoxalmente, muito pouco explorada através de programação genética (Koza, 1992).

Uma forma possível de explorar a programação genética em jogos é através do simulador Robocode (IBM, 2001). Robocode é um simulador de batalhas entre tanques de guerra (IBM, 2001). Cada tanque é controlado por um programa de computador escrito na linguagem Java, informações sobre tal linguagem podem ser obtidas em Flanagam (2000), Haggar (2000) e Newman(1997). Este programa recebe eventos e realiza operações no tanque. Os eventos são a forma que o programa recebe informações sobre o que está ocorrendo na simulação. Eles indicam, por exemplo, se o tanque que o programa está controlando foi atingido pelo inimigo, ou se um disparo que o tanque realizou foi bem sucedido. As operações são a forma como o programa controla as ações do tanque interagindo assim com a simulação. O programa pode, por exemplo, realizar uma operação de "andar para frente" e, na simulação, o tanque irá se mover para frente. Quando dois tanques estão lutando, o confronto, é na verdade, entre dois programas, onde cada programa é executado em uma

thread diferente, ambas controladas pelo simulador. O simulador garante que ambos os programas terão uma quantidade de tempo máximo de processamento, garantindo assim um confronto justo. O simulador Robocode é acompanhado de diversos exemplos de programas controladores de tanques.

Como cada tanque no simulador Robocode é controlado por um programa em Java, pode-se utilizar a programação genética para encontrar tais programas. Como o simulador oferece operações e eventos basta procurar no universo de programas de computador por programas que utilizem tais operações e respondam a tais eventos.

Para realizar busca por programas é necessário desenvolver uma ferramenta capaz de realizar as operações de programação genética. A ferramenta deve testar cada programa controlador de tanques na plataforma de simulação Robocode. A eficiência de cada programa é medida através da pontuação dada a ela pelo simulador. A forma como a ferramenta procura pelos programas é se restringindo somente a programas na linguagem de programação Java que utilizem certas construções da linguagem. Essas construções podem ser operadores condicionais ("if"), operadores de repetição ("while"), valores constantes, variáveis e a chamada de métodos que representam as operações realizáveis no tanque.

1.1 OBJETIVOS DO TRABALHO

Os objetivos principais são:

a) desenvolver o protótipo de uma ferramenta para programação genética;

b) utilizar esse ferramenta para encontrar um programa controlador de tanques no simulador Robocode que seja capaz de derrotar um dos programas de exemplo do simulador chamado "SittingDuck".

1.2 ESTRUTURA DO TRABALHO

O trabalho primeiro descreve o funcionamento da programação genética e do simulador Robocode. Com tais conceitos apresentados é então descrito como foi realizado a implementação e o funcionamento da ferramenta de programação genética que foi desenvolvida. Por último é apresentado os resultados obtidos com a aplicação da ferramenta na busca por um programa controlador de tanque para o simulador Robocode.

2 PROGRAMAÇÃO GENÉTICA

A programação genética foi desenvolvida e exaustivamente pesquisada por John R. Koza sendo seus trabalhos (KOZA 1992, 1994) referência absoluta neste campo, e servem como base para todos os fundamentos de programação genética apresentados neste trabalho.

2.1.1 CONCEITO

Na natureza a sobrevivência dos indivíduos é determinada pela sua capacidade de se adaptar ao meio ambiente em que vive (KOZA, 1992). Essa adaptação é medida pela sua sobrevivência no decorrer do tempo e pela sua capacidade de reprodução. As características de um indivíduo, que o fazem ser adaptado ou não, são descritas pelo seu código genético, seu gene, ou seja, ele é tão adaptado quanto seus genes o permitem ser. Como os indivíduos bem adaptados têm uma tendência a viver mais e reproduzirem mais, seus genes tendem a estar presente em maior quantidade na próxima geração de indivíduos, da mesma forma, os indivíduos mal adaptados tendem a diminuir a presença de seus genes na próxima geração. Eventualmente novos genes serão inseridos na população, seja através da troca de pares de genes (*crossover*) durante o processo de reprodução ou através de mutações aleatórias no próprio gene. Se tais alterações tornarem o individuo mais adaptado então elas provavelmente estarão presentes na próxima geração. A iteração desse processo é conhecida como "seleção natural", cada geração é exposta ao meio ambiente, os bem adaptados sobrevivem, reproduzem e transmitem seus genes para as gerações futuras.

A programação genética utiliza o principio da seleção natural, onde os indivíduos são programas de computador e o meio ambiente é o problema que se deseja resolver.

2.1.2 FUNÇÃO E OBJETIVO

O objetivo da programação genética é encontrar um programa de computador que realize determinada tarefa ou faça determinado calculo.

Para encontrar tal programa o algoritmo de programação genética mantém uma "população" de centenas ou milhares de programas de computador. Cada um desses programas é testado para determinar o quão próximo ele se aproxima do objetivo esperado. Caso em toda a população não se encontre um programa que de a resposta esperada então

uma nova população é gerada tendo como base a população de programas atual. Alguns programas são simplesmente copiados para esta nova população, outros são escolhidos em pares e segmentos dos programas são trocados entre si e então inseridos na nova população. Quanto mais próximo do resultado esperado o programa chegar maior a probabilidade de ele ser selecionado. Após a nova população ter sido formada cada programa é testado e todo o processo se reinicia. A cada nova “geração” os programas tendem a ficaram melhores, e, eventualmente o programa procurado poderá ser encontrado.

Embora a quantidade de programas seja praticamente infinita, a programação genética possui mecanismos para encontrar o programa esperado utilizando-se recursos computacionais limitados.

2.1.3 FUNCIONAMENTO

A programação genética opera um algoritmo bastante simples representado pelo fluxograma da fig. 2.1.

As principais etapas do algoritmo podem ser resumidas em:

a) gerar a população inicial;
b) determinar o *fitness* de cada indivíduo;
c) realizar operações genéticas;
d) checar se condição de término foi atingida.

Existe uma etapa da programação genética que é realizada antes da execução do algoritmo em si. Essa etapa consiste em definir tanto a linguagem de programação quanto o conjunto de funções e terminais que serão utilizados para a geração dos programas.

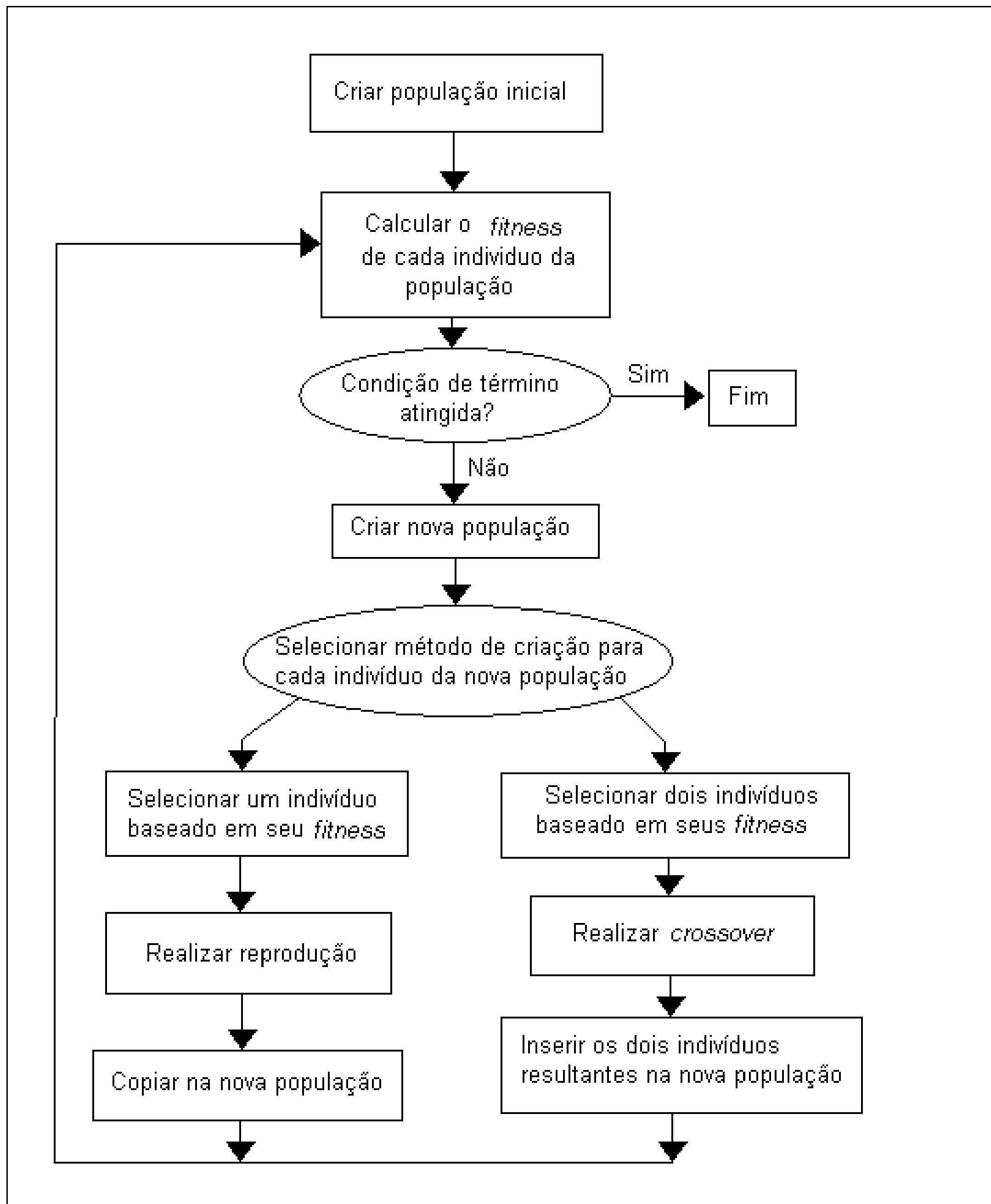


FIGURA 2.1 - Fluxograma da programação genética

2.1.4 LINGUAGEM DE PROGRAMAÇÃO, FUNÇÕES E TERMINAIS

Todo programa de computador pode ser representado através de uma estrutura de dados do tipo árvore, na verdade a maioria dos compiladores, das diversas linguagens de programação existentes, converte internamente o código fonte dos programas para uma árvore composta de funções e terminais durante a compilação. Esse tipo de estrutura é composto por

nós intermediários que dão origem a outros nós e de nós terminais que são as “folhas” das árvores, pois não dão origem a nenhum outro nó, a fig. 2.2 mostra um exemplo de um programa na linguagem de programação Java e sua árvore correspondente, os nós terminais são os círculos e os nós intermediários são as elipses.

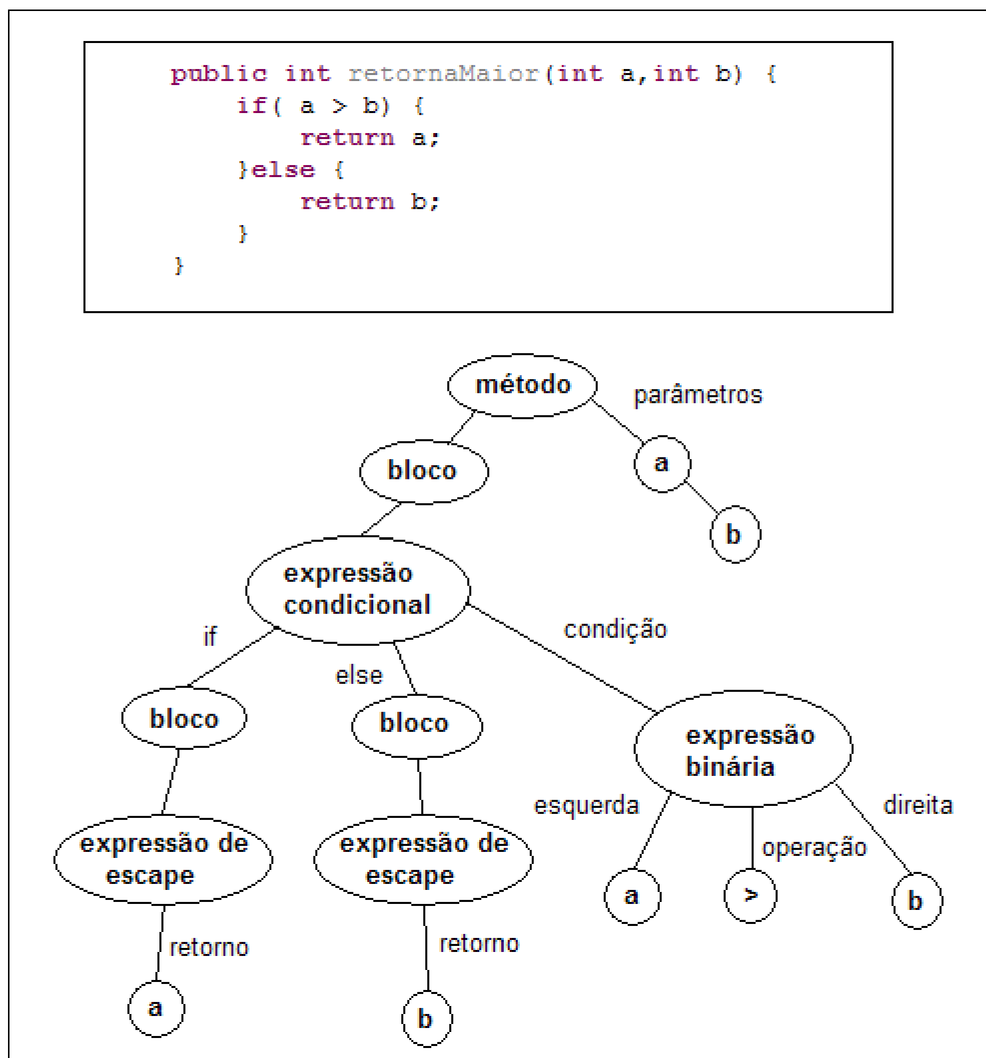


FIGURA 2.2 – Um trecho de código em Java e sua árvore

Para cada problema que se pretende resolver através de programação genética deve selecionar tanto uma linguagem de programação quanto um conjunto de funções e terminais que serão utilizados para montar os programas.

Koza (1992) defende o uso da linguagem de programação LISP, pois a própria estrutura sintática desta linguagem já é descrita como uma árvore, e tal propriedade facilita a

implementação. Não existe qualquer restrição quanto a qual linguagem selecionar, contanto que ela possa ser representada através de uma árvore.

O conjunto de funções e terminas deve ser cuidadosamente escolhido e devem possuir duas propriedades principais, chamadas *closure* e suficiência que serão apresentadas a seguir.

2.1.4.1 PROPRIEDADE CLOSURE

Essa propriedade indica que todos as funções e terminais devem ser compatíveis entre si, ou seja, que o tipo de dado do retorno e dos parâmetros de todas as funções e o tipo de dado dos terminais sejam o mesmo.

Por exemplo, deseja-se obter um programa que tome como entrada um numero inteiro *n* e retorne o n-ésimo número primo a partir de "0" (zero). Tal programa ao ter como entrada o numero inteiro "4" deve retornar o número "7" que é o quarto número primo a partir do zero. Define-se o seguinte conjunto de funções em uma linguagem de programação hipotética:

a) **soma(a, b)** : retorna a soma dos elementos **a** e **b**;

b) **multi(a, b)** : retorna a multiplicação dos elementos **a** e **b**;

c) **seMaior(a, b, c, d)** : se o elemento **a** for maior que o elemento **b** então retorna **c**, senão retorna **d**;

d) **seIgual(a, b, c, d)** : se o elemento **a** for igual ao elemento **b** então retorna **c**, senão retorna **d**.

E os terminais:

a) **INPUT**: O valor inteiro dado como entrada;

b) **CONST_1:** Número inteiro constante com valor "1";

c) **CONST_2:** Número inteiro constante com valor "2".

A propriedade *closure* diz que o tipo de dado do retorno de todas as funções deve ser compatível com o tipo de dado dos parâmetros dessas mesmas funções e que devem ser também compatíveis com o tipo de dado dos terminais. Deve ser possível, portanto, montar um programa válido com qualquer combinação das funções e terminais. A fig. 2.3 mostra a árvore de alguns programas que utilizam o conjunto de funções e terminais apresentado.

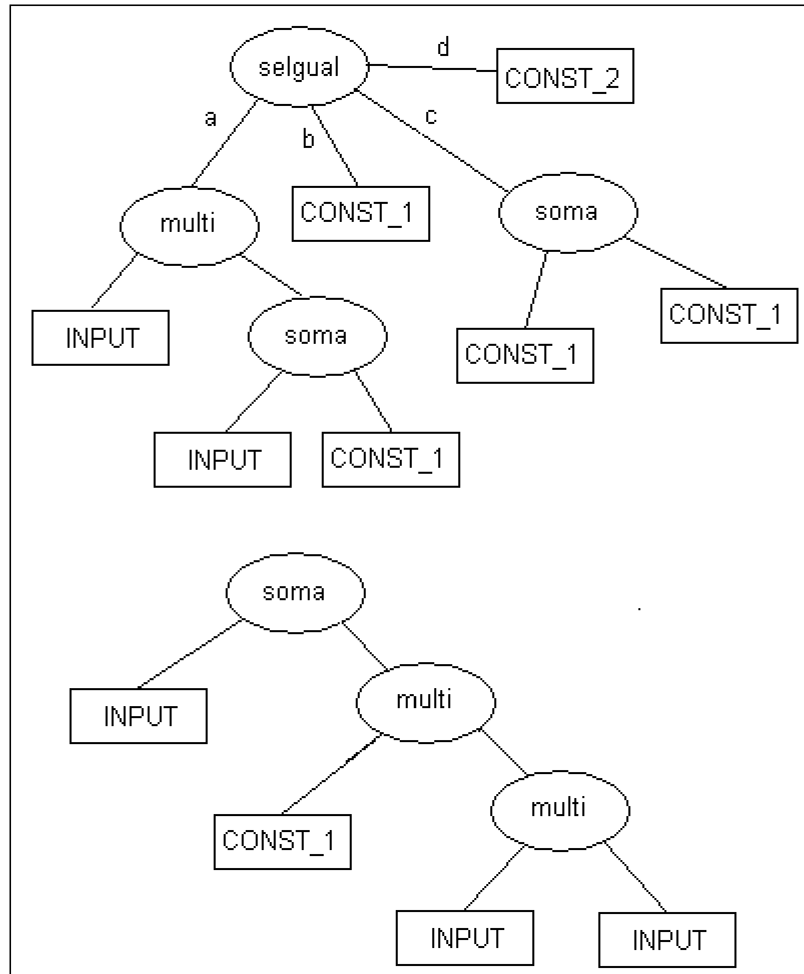


FIGURA 2.3 - Exemplo de programas

Em muitos casos não será possível selecionar um conjunto de funções e terminais que sejam completamente compatíveis devido a alguma restrição sintática da própria linguagem ou por alguma outra peculiaridade do próprio problema. Nestes casos é necessário tomar os devidos cuidados na geração da população inicial e na operação de *crossover* de maneira à sempre gerar um programa sintaticamente válido, dessa maneira a propriedade *closure* é mantida.

2.1.4.2 PROPRIEDADE SUFICIENCIA

A “suficiência” indica que o conjunto de funções e terminais escolhido deve ser capaz de expressar uma solução para o problema proposto, ou seja, uma combinação das funções e terminais deve ser capaz de chegar a uma solução para o problema.

A importância dessa propriedade pode ser expressa através de um exemplo. A fig. 2.4 mostra a equação que retorna o volume de uma esfera. Se fosse tentado obter um programa através de programação genética que dado o raio de uma esfera seja retornado o seu cubo sem fornecer no conjunto das funções a operação de multiplicação ou no conjunto dos terminais o valor de “π” (pi) tal equação jamais seria encontrada.

$$V = \pi \cdot \frac{4}{3} \cdot r^3$$

FIGURA 2.4 – Equação do volume de uma esfera

2.1.4.3 POPULAÇÃO INICIAL

A população inicial é gerada aleatoriamente utilizando-se o conjunto de funções e métodos escolhidos para o problema. Esse processo é bastante simples e direto. Koza (1992) descreve duas abordagens:

a) “crescer”, neste método é escolhido aleatoriamente um nó (função ou terminal) inicial, se tal nó for uma função então escolher aleatoriamente nós para serem seus filhos e assim por diante, profundidade da árvore deve ser limitada por um valor máximo, se tal valor for atingido então se seleciona um terminal ao invés de um nó;

b) “completo”, neste método é determinado uma profundidade máxima para a árvore e a mesma é montada de aleatoriamente, como no método anterior, mas composta somente de funções e que possua a profundidade máxima, então povoar os pontos terminais com nós terminais escolhidos aleatoriamente.

2.1.5 AVALIAÇÃO DOS PROGRAMAS - FITNESS

A medida da adaptação de um individuo ao seu meio é, em programação genética, denominada *fitness*. A medição do *fitness* é única para todos os indivíduos de uma população e varia de acordo com o problema em questão.

2.1.5.1 RAW FITNESS

É a medida bruta da avaliação do individuo, ou seja, o resultado direto do programa que representa o indivíduo. Essa medida é completamente dependente do problema em si, mas deve ser um valor numérico, inteiro ou real. Se o resultado direto do problema não for um valor numérico deve-se avaliar cada individuo da população e determinar o *fitness* como uma escala entre o melhor e o pior. Exemplos de *raw fitness* são exibidos na tabela 2.1.

Tipo do problema	*Raw fitness*
Menor caminho entre 2 pontos de grafo	A quantidade de nós visitados entre os dois pontos.
Regressão simbólica de *n* pontos em um gráfico.	A somatória do "erro" dos *n* pontos do programa gerado. O "erro" é a diferença entre a coordenada *e* esperada e a coordenada *g* gerada pelo programa.
Descobrir a quantidade de números primos entre 0 e *n*.	Um número inteiro retornado pelo programa.

TABELA 2.1 –Exemplos de *raw fitness.*

Em alguns casos o *raw fitness* poderá ser uma medida de erro, ou seja, quanto maior pior, ou uma medida de sucesso, quanto maior melhor.

2.1.5.2 STANDARIZED FITNESS

É idêntica a *raw fitness,* mas com a garantia de que quanto menor o valor numérico mais adaptado é o individuo, e onde o individuo ideal possui *standarized fitness* igual a zero.

Para casos onde o *raw fitness* é uma medida de erro, ambos *standarized fitness* e *raw fitness* terão valor idêntico, exceto em casos que o *raw fitness* nunca atinja o valor zero, nestes casos é subtraído um valor constante da *raw fitness.* Esse valor constante corresponde ao valor mínimo que o *raw fitness* pode atingir, dessa forma o *standarized fitness* é capaz de atingir zero.

Para casos onde o *raw fitness* é uma medida de sucesso, ou deve-se computar o *standarized fitness* subtraindo-se o valor de *raw fitness* de um valor constante. Esse valor constante é o valor máximo que o *raw fitness* é capaz de atingir.

2.1.5.3 ADJUSTED FITNESS

Tem como base a *standarized fitness* e seu valor sempre estará no intervalo [0,1]. É computada através da fórmula dada na fig. 2.5, onde "s" é o valor da *standarized fitness*.

$$a = \frac{1}{1 + s}$$

FIGURA 2.5 – Formula do *adjusted fitness*

O *adjusted fitness* tem a característica de exagerar a importância de pequenas diferenças do *standarized fitness* quando este se aproxima de zero. Por exemplo, em determinado problema o valor do *standarized fitness* varia no intervalo [0,64] onde 0 é melhor e 64 é pior. O valor de *adjusted fitness* de dois indivíduos "ruins" com *standarized fitness* de 64 e 63 serão respectivamente 0,0154 e 0,0159, já o *adjusted fitness* de dois indivíduos "bons" com *standarized fitness* de 4 e 3 serão respectivamente de 0,20 e 0,25.

2.1.5.4 NORMALIZED FITNESS

Esse método atribui a cada individuo um valor de *fitness* correspondente a sua proporção em relação a todos os outros indivíduos da população. O *normalized fitness* é calculado através da formula apresentada na fig. 2.6, onde "a" é o valor da *adjusted fitness* do indivíduo, "M" o total de indivíduos.

$$n = \frac{a}{\sum_{k=1}^{M} a_k}$$

FIGURA 2.6 – Formula da *normalized fitness*

A *normalized fitness* tem três características desejáveis :

a) varia entre 0 e 1;
b) possui valor maior para melhores indivíduos;
c) a soma da *normalized fitness* de todos os indivíduos da população é igual a 1;

2.1.6 OPERAÇOES GENETICAS

Cada nova geração de indivíduos é gerada aplicando-se duas operações genéticas na população anterior. As principais operações genéticas descritas por Koza (1992) são:

a) reprodução Darwiniana;
b) crossover (recombinação sexuada).

Cada individuo da nova geração é o resultado de uma dessas duas operações, a escolha de qual operação é dada através de uma probabilidade. As probabilidades devem, obrigatoriamente somar 100%. Por exemplo, definimos que a probabilidade de reprodução é de 80% e a de *crossover* é de 20%, esses valor são uma probabilidade e como resultado aproximadamente 80% da nova população será gerada através de reprodução e 20% através de *crossover.*

Ambas operações são descritas com detalhes nas próximas seções.

2.1.6.1 REPRODUÇÃO

É a força base da seleção natural Darwiniana. Opera em somente um individuo e gera como resultado somente um individuo.

A operação de reprodução consiste em duas etapas. Primeiro um individuo da população é escolhido através de algum método de seleção baseado em sua *fitness.* Segundo,

o individuo selecionado é inserido, sem qualquer alteração, da população atual para a nova população.

Existem vários métodos de seleção, segundo Koza (1992) o método mais popular é a seleção proporcional a *fitness.* Neste método o individuo é escolhido com uma probabilidade igual ao seu *normalized fitness.* Para melhor visualizar o funcionamento deste método pode-se imaginar uma roleta como na fig. 2.7, onde os números correspondem ao *fitness*, e a porcentagem entre parênteses corresponde a sua participação na "roleta". Os indivíduos são selecionados "rolando-se" a roleta.

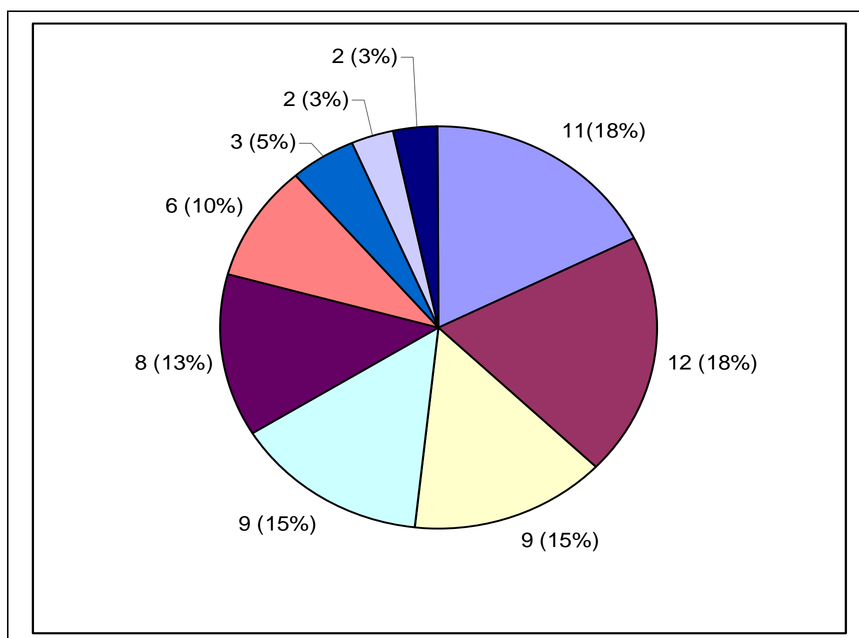


FIGURA 2.7 – "Roleta" de uma seleção proporcional a *fitness*

Entre outros métodos os principais são por *ranking* e por torneio. No método de *ranking* os indivíduos são selecionados tomando como base a colocação do individuo na escala de *fitness* e não no próprio valor do mesmo. No método de torneio dois indivíduos são selecionados aleatoriamente da população e aquele com melhor *fitness* é inserido na nova população.

2.1.6.2 CROSSOVER

A operação de *crossover* aumenta a variedade genética na população ao inserir novos elementos na mesma. Opera em dois indivíduos e gera como resultado dois novos indivíduos.

Essa operação realiza a troca, entre os dois indivíduos envolvidos, de fragmentos da árvore dos programas de ambos indivíduos, esse processo é realizado em três etapas básicas.

a) Dois indivíduos são selecionados na população atual independentemente, através do mesmo método utilizado para a seleção dos indivíduos para a operação de reprodução;
b) É selecionado um nó aleatório na árvore dos programas de ambos indivíduos. A seleção é puramente aleatória e pode ser feita numerando os nós e escolhendo um dos números através de algum método de geração de números aleatórios. Esses nós são a raiz de uma sub-árvore no programa destes indivíduos;
c) Os indivíduos trocam entre si essas sub-árvores.

A figura 2.8 mostra duas árvores A e B contendo as funções e terminais apresentados no cap. 2.1.4.1 e indica com uma coloração cinza clara o nó escolhido para *crossover* na árvore A e uma coloração cinza escura o nó escolhido para o *crossover* na árvore B, a fig. 2.9 mostra as duas árvores após a operação.

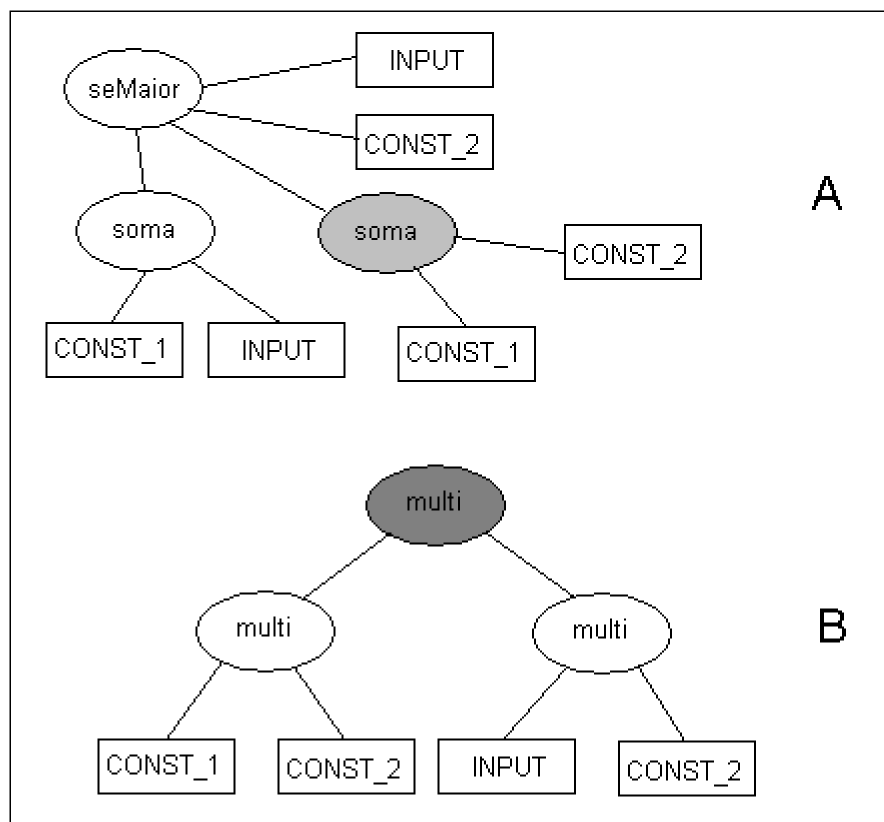


FIGURA 2.8 – Árvores e nós escolhidos para *crossover*

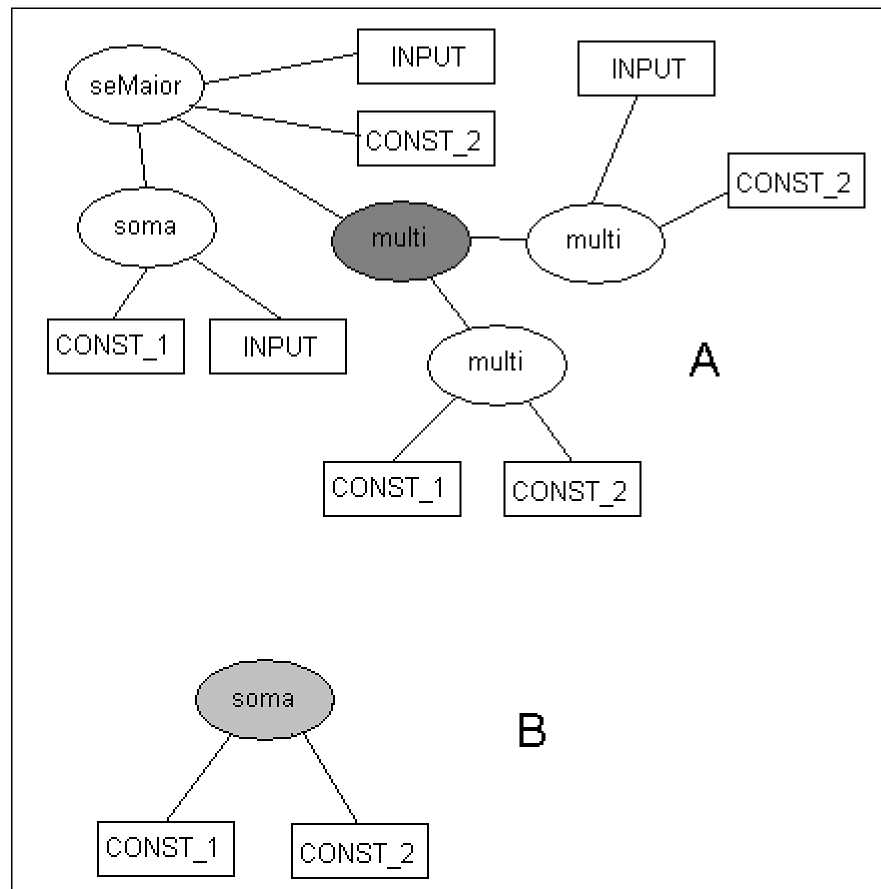


FIGURA 2.9 – Árvores após o *crossover*

Devido à propriedade c*losure* das funções e terminais essa troca de sub-árvores sempre gera um programa válido, independente da seleção dos indivíduos ou dos nós escolhidos para a operação.

O nó escolhido na árvore B foi sua própria raiz, portanto toda a árvore de B foi substituída pelo fragmento de árvore de A. Segundo Koza (1992) esse tipo de situação peculiar pode acontecer de diversas maneiras:

Se o nó escolhido em um dos indivíduos é um terminal então esse terminal é substituído pela sub-árvore do outro individuo, e esse ultimo recebe o nó terminal no lugar de sua sub-árvore. Esse processo é mostrado na fig. 2.8 onde um terminal, em cinza, da árvore A é substituído por uma função, em amarelo da árvore B.

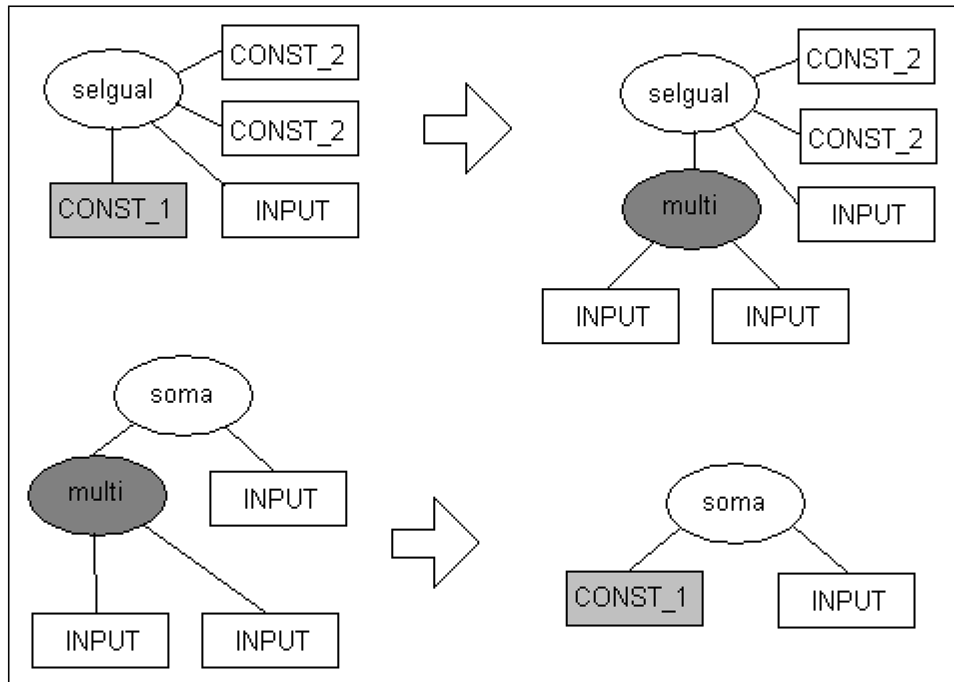


FIGURA 2.10 - *Crossover* entre terminal e sub-árvore

Se os nós escolhidos de ambos os indivíduos forem nós terminais então eles são simplesmente trocados de uma árvore para outra. A fig. 2.9 exibe duas árvores A e B que tem seus nós terminais trocados, os terminais estão marcados em cinza claro e cinza escuro.

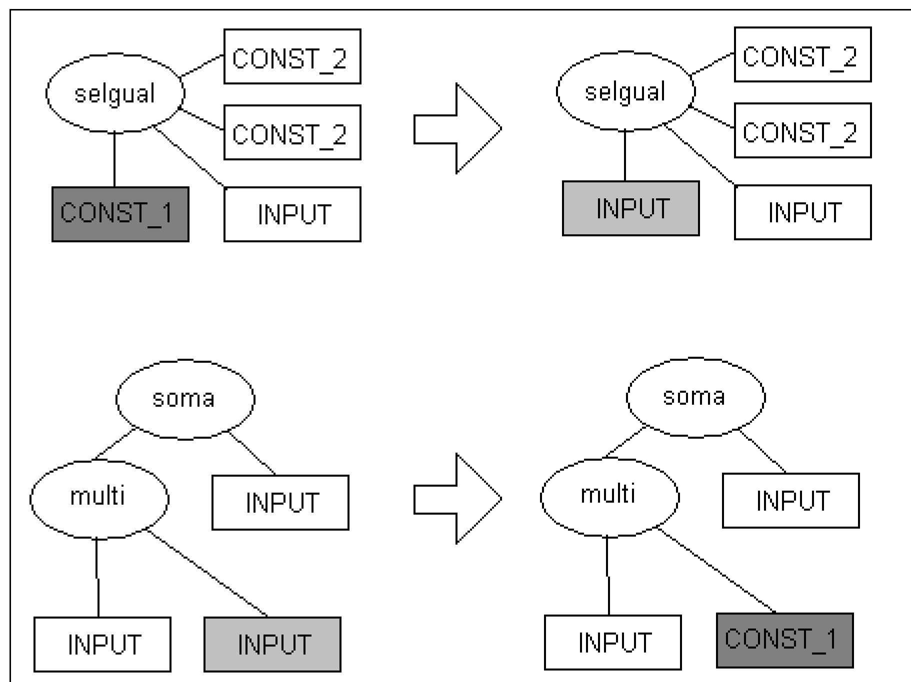


FIGURA 2.11 - *Crossover* entre terminal e terminal

Se a raiz do programa de um indivíduo for selecionado então toda a árvore deste indivíduo será inserida no nó escolhido do outro indivíduo, como foi mostrado nas fig. 2.6 e 2.7.

Se, como no caso anterior, a raiz de um individuo for selecionada e o nó selecionado do outro indivíduo for um terminal então o primeiro indivíduo passa a ser um programa composto de somente um terminal, mostrado na fig. 2.10.

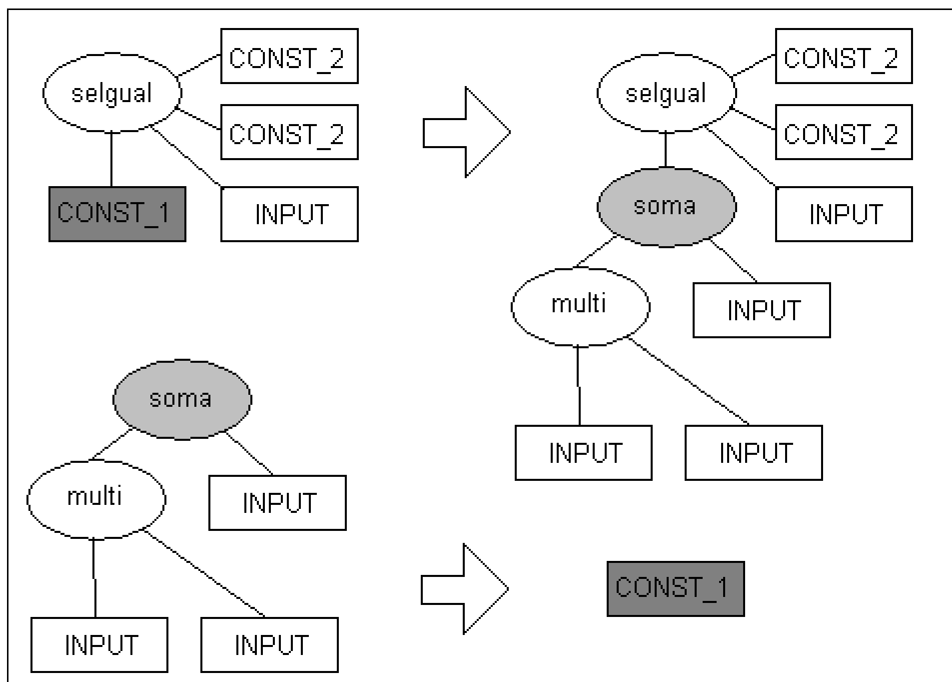


FIGURA 2.12 - *Crossover* entre raiz e terminal

Se ambos os nós escolhidos forem às raízes dos programas então ambos os indivíduos são inseridos na nova população sem qualquer alteração.

Se os dois indivíduos selecionados forem o mesmo, ou tiverem uma árvore idêntica, então o resultado será dois indivíduos distintos somente se os nós escolhidos forem diferentes. Se os nós escolhidos forem os mesmos, então, como no caso anterior, os indivíduos serão inseridos inalterados na nova população.

A operação de *crossover* costuma gerar indivíduos com árvores de profundidade muito grande, por esse motivo é estabelecido um limite para a profundidade das árvores. Se a arvore final de qualquer um dos indivíduos for maior que o máximo permitido a operação é cancelada para este indivíduo e ele é inserido inalterado na nova população, a operação

continua válida para o outro indivíduo. Se as novas árvores de ambos indivíduos excederem o tamanho máximo, ambos são inseridos na nova população inalterados.

2.1.7 CONDIÇÃO DE TÉRMINO

A programação genética é um processo infinito, novas gerações podem ser geradas indefinidamente, portanto o processo deve ser interrompido de alguma maneira. Segundo Koza (1992) há duas maneiras de fazer isso.

Uma delas é estabelecer um limite para o número de gerações. Esse método é geralmente utilizado para problemas que não se espera um resultado exato, tais como ajustes de curvas, ou para problemas que não se sabe determinar o resultado final.

A outra maneira é encontrar um individuo que gere uma solução 100% correta. Esse método é utilizado em problemas que possuam uma resposta exata e consegue-se identificar tal resposta quando um indivíduo a gera.

2.1.8 RESULTADO DA PROGRAMAÇÃO GENÉTICA

O resultado do algoritmo de programação genética é o melhor indivíduo encontrado em todas as gerações. Esse indivíduo é mantido a parte da população, pois não há garantias que um indivíduo melhor apareça na geração seguinte, pode ocorrer o contrario, o melhor indivíduo da população n+1 ser pior que o melhor indivíduo da população n (KOZA, 1992). Caso em determinada geração apareça um indivíduo melhor do que o melhor indivíduo encontrado até o momento, esse ultimo é substituído pelo primeiro e mantido separado da população.

2.1.9 TRABALHOS CORRELATOS

3 ROBOCODE

Robocode é um jogo, um simulador de batalhas entre tanques de guerra. Foi desenvolvido pela IBM como uma ferramenta para o aprendizado da linguagem de programação Java (IBM, 2001).

3.1.1 FUNCIONAMENTO

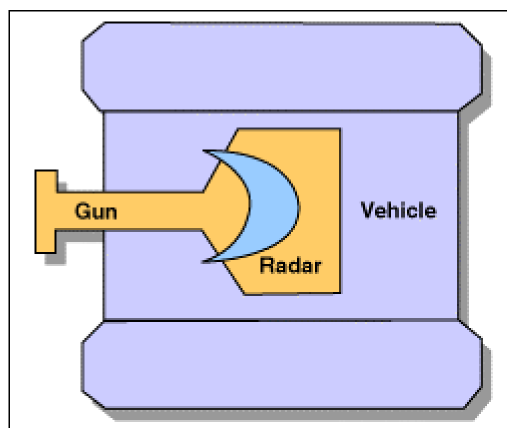
Cada tanque de guerra no simulador robocode é controlado por um programa na linguagem Java. Esse programa recebe eventos do simulador e realiza operações em resposta a estes eventos.

3.1.2 ANATOMIA DO TANQUE

Um tanque é composto por três componentes básicos:

a) corpo: responsável pelo movimento;
b) canhão: realiza disparos;
c) radar: localiza inimigos.

A fig. 3.1 mostra localização desses elementos no tanque.



Fonte: Li (2002)

FIGURA 3.1 – Anatomia de uma tanque no Robocode. Onde *Gun* é o canhão, *Vehicle* é o corpo e *Radar* é o próprio radar.

O canhão está montado em cima do corpo e o radar está montando em cima do canhão. Mas nem o canhão é fixo no corpo, nem o radar é fixo no canhão. Quando o corpo gira o canhão e o radar podem acompanhar o movimento ou não, esse comportamento é “configurável” e o programa que controla o tanque pode optar por um comportamento ou outro.

As principais operações que se pode realizar no corpo do tanque são:

d) mover para frente;
e) mover para a trás;
f) girar para a esquerda;
g) girar para a direita;
h) parar.

As principais operações com o canhão são:

a) disparar;
b) girar para a esquerda;
c) girar para a direita.

As principais operações com o radar são:

a) girar para a esquerda;
b) girar para a direita;
c) procurar por inimigos.

Os principais eventos enviados pelo simulador são:

d) Quando o tanque é atingido por um disparo;
e) Quando o tanque atinge outro tanque;
f) Quando o tanque colide com outro tanque;
g) Quando o tanque colide com uma parede;
h) Quando o radar detecta um tanque.

3.1.3 REGRAS DO JOGO

Cada tanque inicia o combate com 100 pontos de energia. No decorrer da batalha esses pontos podem ser perdidos ou mais pontos podem ser ganhos.

Um tanque pode perder pontos de energia da seguinte maneira :

a) ao ser atingido por uma disparo perde-se uma quantidade de energia equivalente a 4 vezes a potência do disparo, se a potência for maior ou igual a 2 então é realizado um dano adicional de 2*(potência -1);

b) ao realizar um disparo perde-se uma quantidade de energia equivalente à potência do disparo realizado. Tal potência é um valor real que pode variar entre 1 e 3;

c) ao colidir com outro tanque de qualquer ângulo ambos perdem determinada quantia de energia enquanto, o autor não foi conseguiu determinar como tal quantidade é calculada pelo simulador;

E ganha pontos de energia em tais eventos :

d) ao atingir outro tanque com um disparo do canhão se ganha 3 vezes a potência do disparo.

Quando restar somente 1 ponto de energia o tanque fica imobilizado e não é mais capaz de realizar qualquer ação, o programa controlador não é mais executado pelo simulador. A única escapatória deste estado é que algum disparo do canhão realizado antes da imobilização atinja outro tanque, dessa forma alguns pontos de energia seriam recuperados e o tanque voltaria à atividade, ou seja, seu programa recomeçaria a ser executado pelo simulador, se isso não acontecer o tanque permanecerá imóvel até o fim da batalha ou até ser destruído.

Ao atingir 0 (zero) pontos de energia o tanque é destruído e seu programa é encerrado.

3.1.4 CAMPO DE BATALHA

As batalhas no Robocode são realizadas em campo de batalha virtual. As dimensões deste campo podem ser definidas pelo usuário e são medidas em *pixel,* o padrão do simulador é uma resolução de 800 linhas por 600 colunas.

O campo é um simples plano delimitado por paredes fixas e indestrutíveis. Não há obstáculos. A fig. 3.2 exibe um campo de batalha típico com dois tanques duelando.

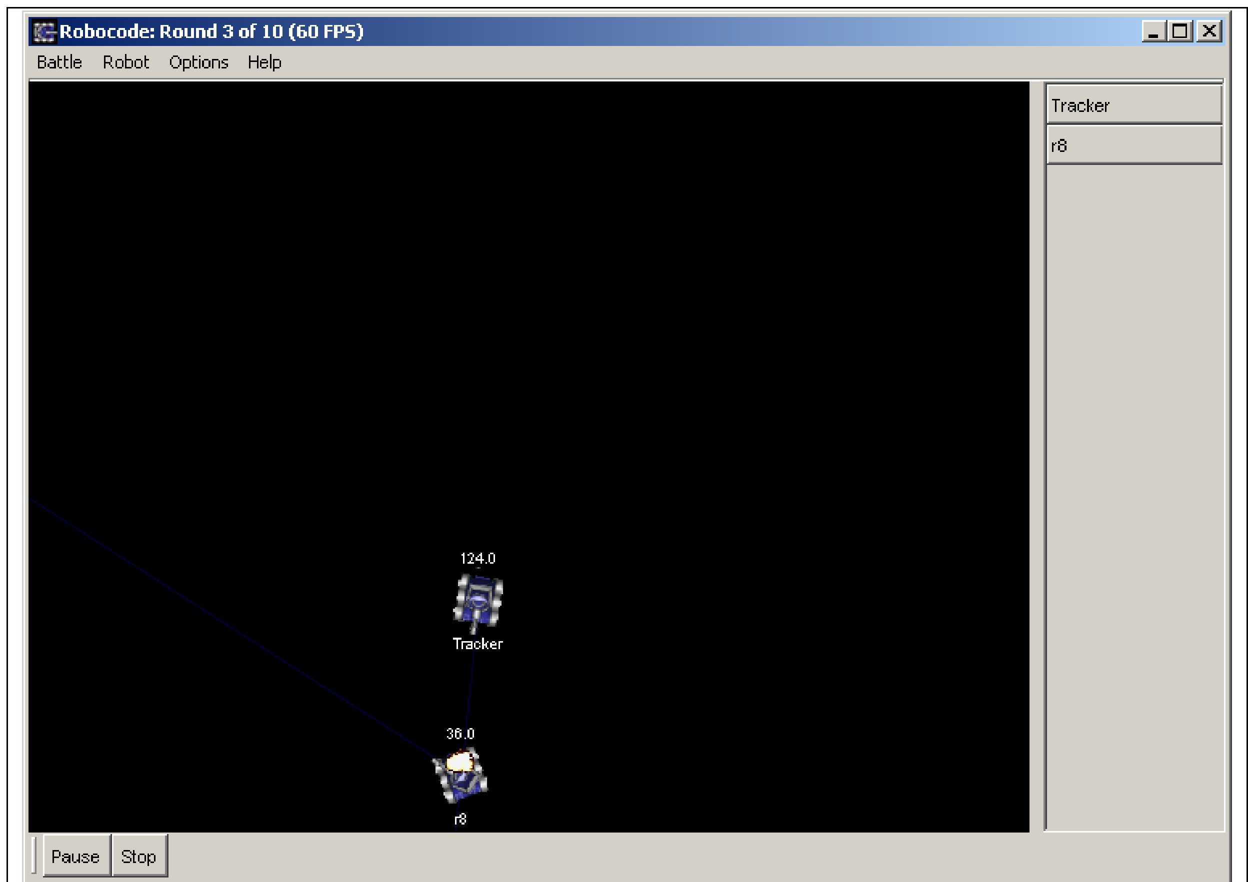


FIGURA 3.2 – Dois tanques “duelando” no simulador Robocode.

3.1.5 BATALHA E PONTUAÇÃO

Pode-se realizar batalhas com diversos tanques simultaneamente, mas toda batalha somente chega ao fim quando restar um único tanque. O vencedor da batalha é determinado através de um sistema de pontuação. São atribuídos pontos da seguinte maneira:

e) sobrevivência: sempre que um tanque é destruído todos os tanques que ainda estão na batalha recebem 50 pontos.

f) sobrevivente: o último tanque restante recebe um bônus de 10 pontos para cada tanque que foi destruído, independente se tais tanques tenham sido destruído pelo sobrevivente ou não.

g) danos por tiro: cada tanque recebe 1 ponto para cada 1 ponto de dano que um disparo por ele realizado tenha ocasionado em outro tanque.

h) Danos por colisão: cada 1 ponto de dano causado em outro robô através de um colisão é recebido 2 pontos.

A fig. 3.3 exibe uma grande batalha entre vários tanques.

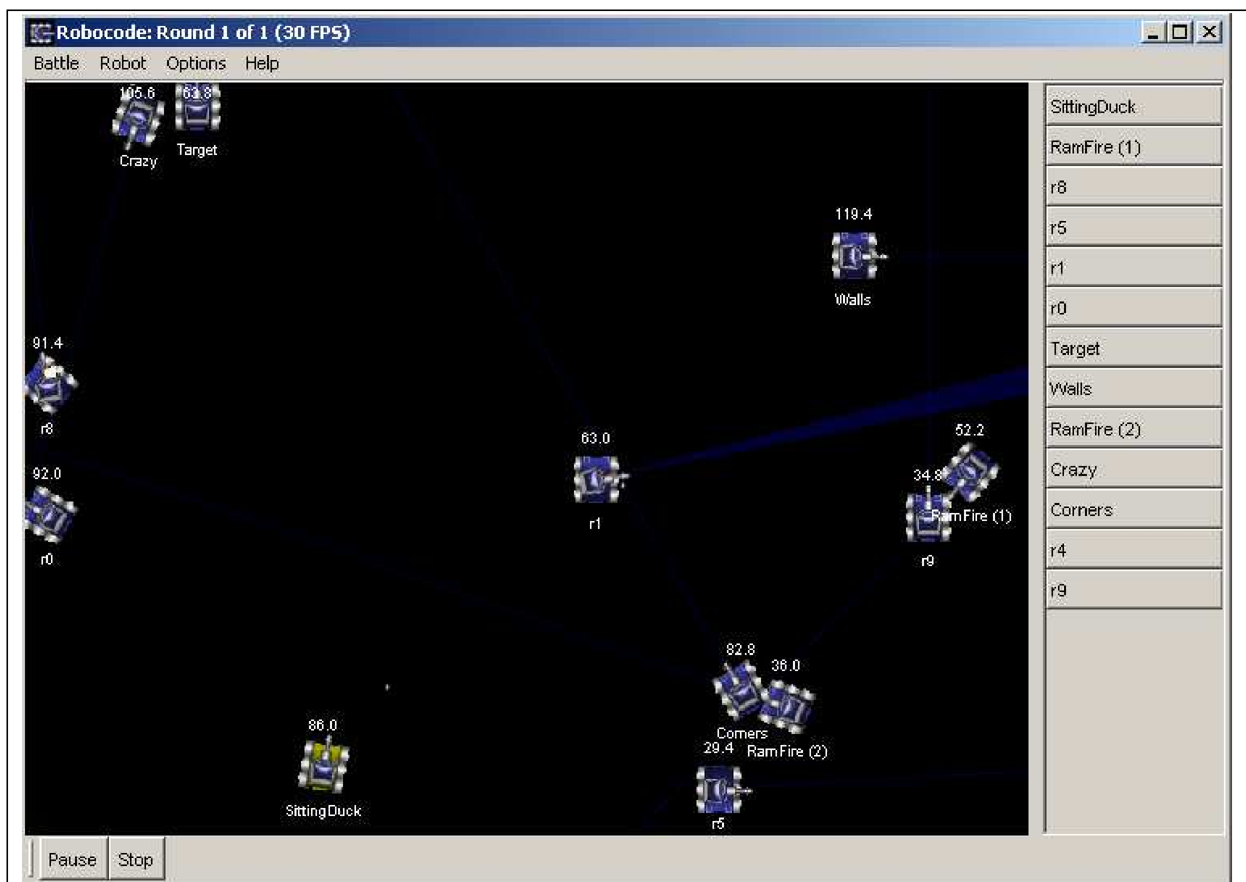


FIGURA 3.3 – Uma batalha envolvendo vários tanques.

Como a posição inicial de cada tanque é aleatória geralmente as batalhas são repetidas algumas vezes para eliminar a possibilidade de determinado tanque ter vantagem devido a um posicionamento inicial favorável.

3.1.6 ESTRUTURA DO PROGRAMA CONTROLADOR

Os programas controladores de tanques são classes escritas na linguagem de programação Java. O simulador Robocode disponibiliza uma classe chamada *Robot* que as classes (programas) controladores devem derivar.

A classe *Robot* possui alguns métodos que são as operações que podem ser realizadas com o tanque. Se a classe controladora deseja realizar alguma operação com o tanque é necessário invocar um desses métodos. Os métodos para as principais operações mostradas no cap. 3.1.2 são:

a) mover tanque para frente:
   - `void ahead(double distance);`

b) mover para a trás:
   - `void back(double distance)`

c) girar tanque para a esquerda:
   - `void turnLeft(double degrees)`

d) girar tanque para a direita:
   - `void turnRight(double degrees)`

e) parar:
   - `void stop()`

f) disparar:
   - `void fire(double power)`

g) girar canhão para a esquerda:
   - `void turnGunLeft(double degrees)`

h) girar canhão para a direita:
   - `void turnGunRight(double degrees)`

i) girar radar para a esquerda:
   - `void turnRadarLeft(double degrees)`

j) girar radar para a direita:
   - `void turnRadarRight(double degrees)`

k) procurar por inimigos:
   - `void scan()`

A classe *Robot* também oferece métodos que só possuem funcionalidade se forem sobrecarregados, esses métodos são os eventos. Ao acontecer determinado evento o simulador

invoca o método correspondente no objeto da classe controladora do tanque, se a classe não deriva tal método então ela não terá conhecimento do evento. Os métodos para os eventos mostrados no cap 3.1.2 são:

a) Quando o tanque é atingido por um disparo:
   - `void onHitByBullet(HitByBulletEvent event)`

b) Quando um disparo do tanque atinge outro tanque:
   - `void onBulletHit(BulletHitEvent event)`

c) Quando o tanque colide com outro tanque:
   - `void onHitRobot(HitRobotEvent event)`

d) Quando o tanque colide com uma parede;
   - `void onHitWall(HitWallEvent event)`

e) Quando o radar detecta um tanque.
   - `void onScannedRobot(ScannedRobotEvent event)`

Entre os eventos há um que é especial, esse método é invocado uma única vez, quando o objeto da classe controladora é instanciado.

```
void run()
```

3.1.7 EXEMPLO DE ROBO

A fig. 3.4 exibe o código fonte do tanque “Fire” que acompanha o simulador Robocode, e a fig. 3.5 exibe o código fonte do tanque “Crazy” que também acompanha o simulador Robocode, os comentários de ambos os códigos fontes foram removidos por serem muito extensos.

```
package sample;
import robocode.*;

public class Fire extends Robot {
    int dist = 50;

    public void run() {
        while (true) {
            turnGunRight(5);
        }
    }

    public void onScannedRobot(ScannedRobotEvent e) {
        if (e.getDistance() < 50 && getEnergy() > 50)
            fire(3);
        else
            fire(1);
        scan();
    }

    public void onHitByBullet(HitByBulletEvent e) {
        turnRight(normalRelativeAngle(90 - (getHeading()
                  - e.getHeading())));
        ahead(dist);
        dist *= -1;
        scan();
    }

    public void onHitRobot(HitRobotEvent e) {
        double turnGunAmt = normalRelativeAngle(e.getBearing()
                + getHeading() - getGunHeading());
        turnGunRight(turnGunAmt);
        fire(3);
    }


    public double normalRelativeAngle(double angle) {
        if (angle > -180 && angle <= 180)
            return angle;
        double fixedAngle = angle;
        while (fixedAngle <= -180)
            fixedAngle += 360;
        while (fixedAngle > 180)
            fixedAngle -= 360;
        return fixedAngle;
    }
}
```

FIGURA 3.4 – Código fonte do tanque “Fire”

```java
package sample;
import robocode.*;

public class SpinBot extends AdvancedRobot {

    public void run() {
        while (true) {
            setTurnRight(10000);
            setMaxVelocity(5);
            ahead(10000);
        }
    }

    public void onScannedRobot(ScannedRobotEvent e) {
        fire(3);
    }

    public void onHitRobot(HitRobotEvent e) {
        if (e.getBearing() > -10 && e.getBearing() < 10)
            fire(3);
        if (e.isMyFault())
            turnRight(10);
    }
}
```

FIGURA 3.5 – Código fonte do tanque “Crazy”

4 ESPECIFICAÇÃO DO SISTEMA

Neste capitulo apresenta-se a especificação do protótipo desenvolvido. São descritas as especificações da ferramenta os detalhes sobre sua implementação, características e a forma de utilização.

4.1 REQUISITOS DO SISTEMA

O objetivo do ferramenta é a realização da programação genética e utilizá-la com o simulador de tanques Robocode.

Para atingir tal objetivo a ferramenta deve ser capaz de:

a) realizar as duas operações básicas da programação genética, a reprodução e o *crossover,* utilizando-se funções e terminais da linguagem de programação Java;

b) avaliar os indivíduos da população.

O protótipo foi batizado de “Jnetic”, uma referência a programação genética utilizando Java.

4.2 ESPECIFICAÇÃO DA FERRAMENTA JNETIC

A análise da ferramenta foi feita através do paradigma de orientação a objetos (AMBLER, 1997). A modelagem das classes foi realizada através da UML (*Unified Modelling Language*) utilizando-se a ferramenta de PoseidonUML (Gentleware, 2003). Informações sobre UML podem ser obtidas em Booch (2000).

O sistema é composto pelas seguintes classes:

a) Engine: É a classe principal, concentra diversas outras classes para a execução do programa. É nessa classe que está o *loop* de gerações da programação genética;

b) XMLLoader: É responsável pela leitura do arquivo XML de configuração e sua interpretação para uma estrutura de dados interna. Com as informações obtidas do arquivo instancia objetos das classes “IGeneticDescriptor”, “IIndividualDescriptor”, “IJavaSourceFile” e “JavaContext”;

c) Evaluator: Uma classe abstrata que o usuário do protótipo deve herdar. É responsável pela avaliação da *fitness* dos indivíduos;

d) Reproducer: Realiza a operação genética de reprodução;

e) Crossover: Realiza a operação genética de *crossover*;

f) Population: Representa uma população de indivíduos, oferece métodos para a manipulação dos indivíduos e informações sobre os mesmos como um todo. Também é capaz de gerar uma população aleatória.

g) JavaContext: Representa um conjunto de funções e terminais Java, oferece métodos para a obtenção de nós ou terminais baseado em seu tipo de retorno.

h) ClassMethod: Representa um contexto Java, contém a declaração de um método e um conjunto de nós e terminais.

i) ClassGenerator: Responsável pela compilação dos indivíduos para classes executáveis Java. Utiliza um objeto “SourceGenerator” para gerar os arquivos fonte, e então os compila utilizando o compilador fornecido com a plataforma Java.

j) SourceGenerator: Transforma a árvore de um individuo em um arquivo fonte Java.

k) JavaIndividual: Representa um indivíduo da programação genética. Possui diversos métodos para a manipulação e a obtenção de informações sobre sua árvore de programa.

l) IJavaSourceFile: Contém informações que descrevem um arquivo fonte Java, utilizado pela classe “SourceGenerator” para auxilio na geração dos arquivos fonte.

m) IIndividualDescriptor: Contém informações especificas sobre as características dos indivíduos.

n) IGeneticDesciptor: Contém informações sobre os parâmetros de execução da programação genética.

A fig. 4.1 exibe o diagrama de classes UML da estrutura geral do sistema, como o diagrama é muito extenso não são apresentados os métodos e atributos, mas somente a relação entre as classes.

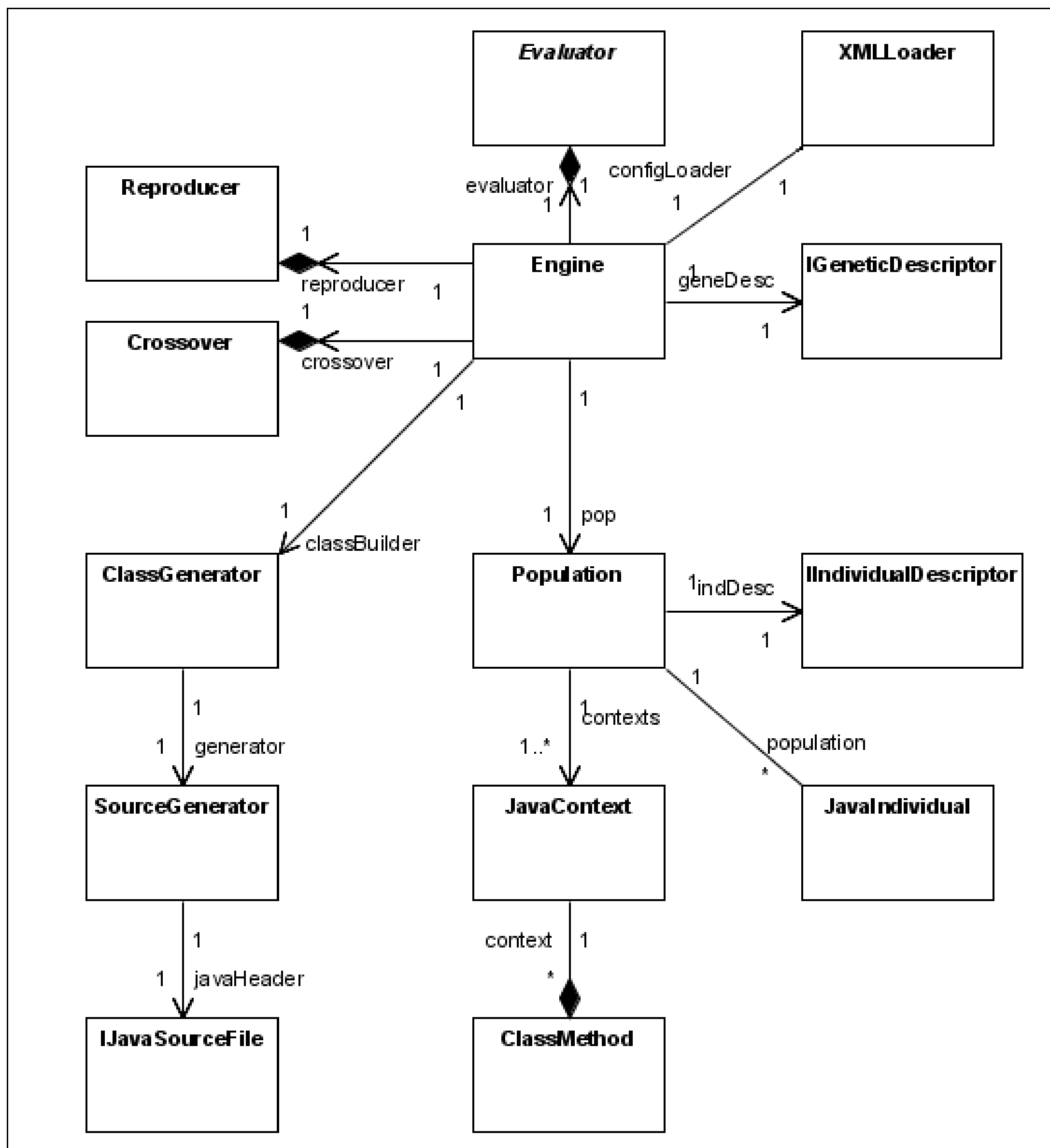


FIGURA 4.1 – Diagrama de classes

O detalhamento é apresentados em três outras figuras, a fig. 4.2 contendo um diagrama parcial do sistema focado na classe “Engine” e sua relação com outras classes:

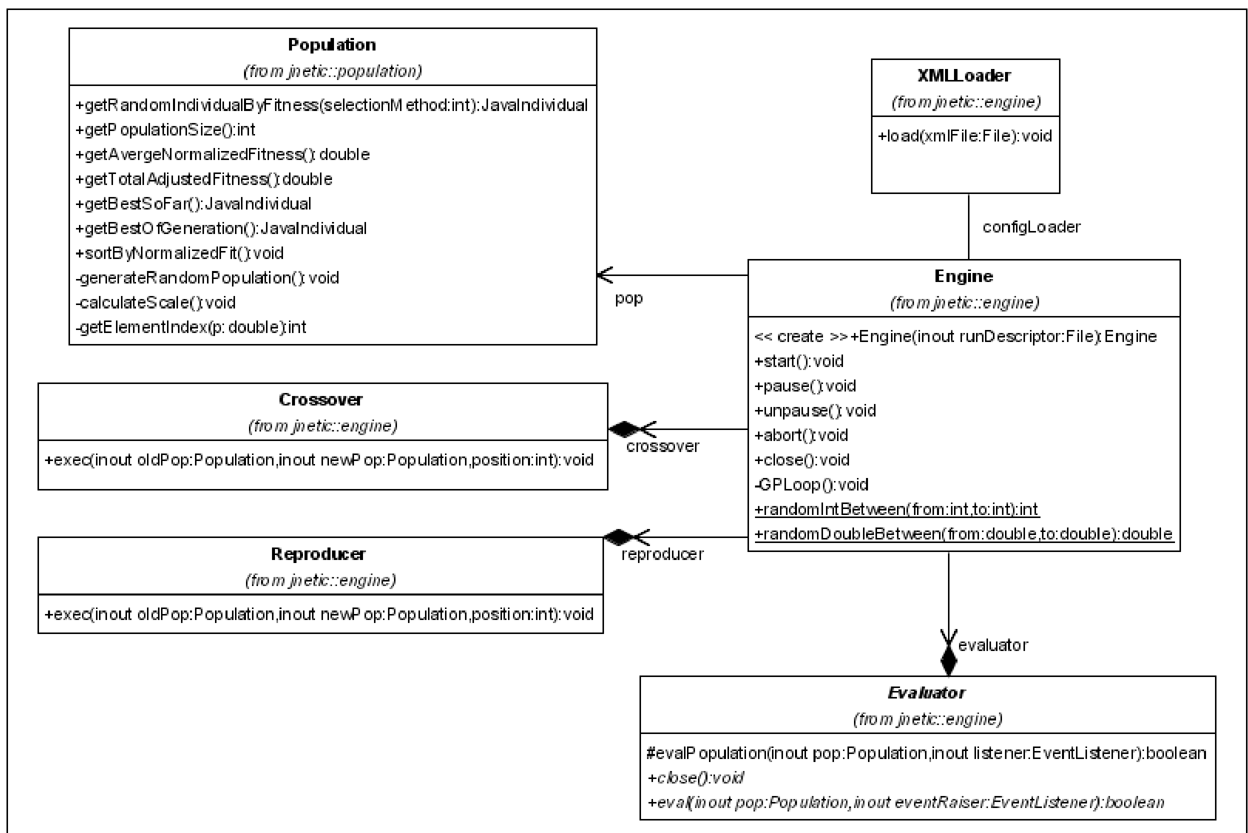


FIGURA 4.2 – Fragmento do diagrama de classes UML centrado na classe Engine

A fig. 4.3 contendo um diagrama parcial centrado na classe “Population” exibindo as classes com as quais possui algum tipo de associação:

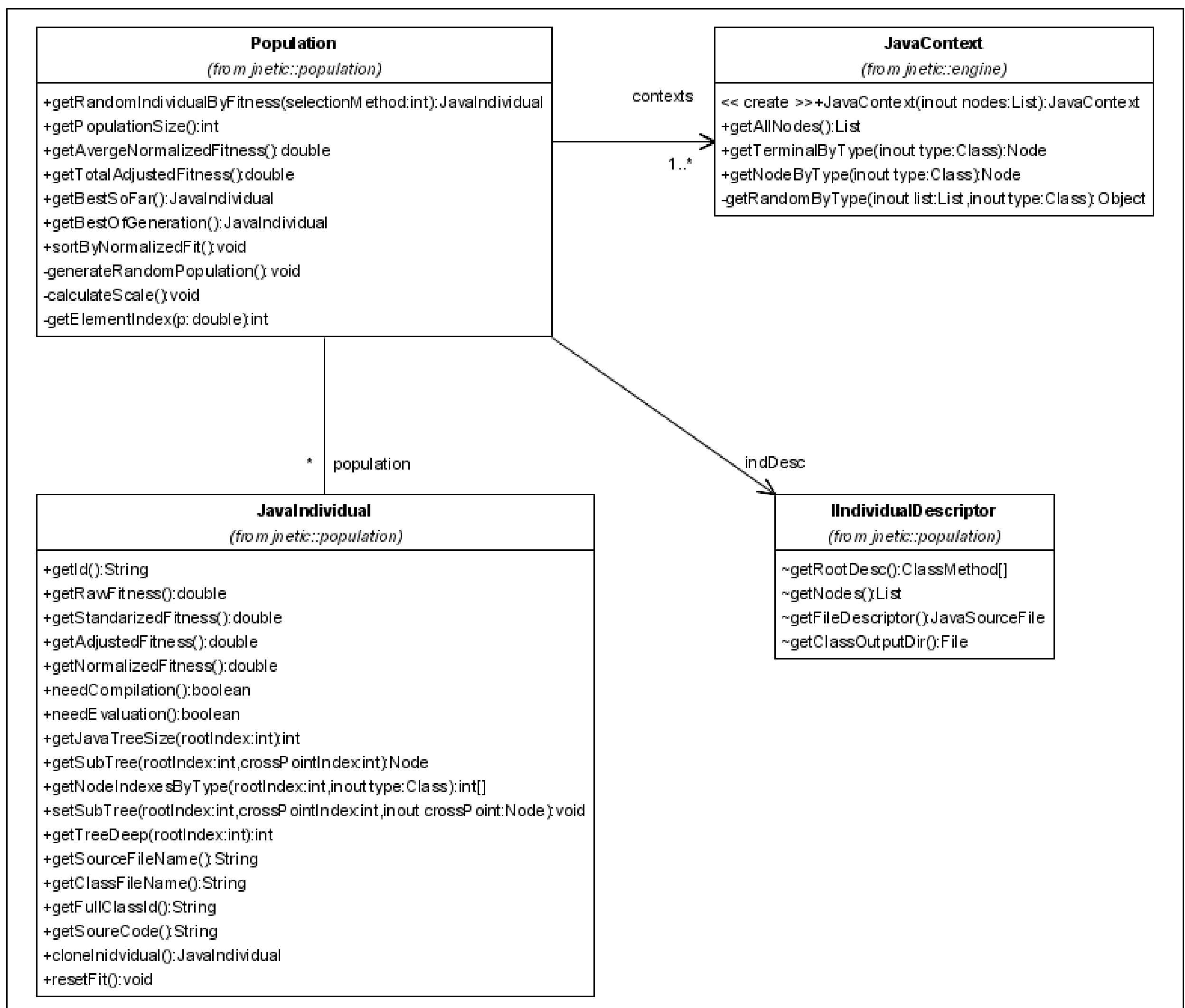


FIGURA 4.3 – Fragmento do diagrama de classes UML centrado na classe Population

A fig. 4.4 contendo um diagrama parcial centrado na classe “ClassGenerator” e suas associações:

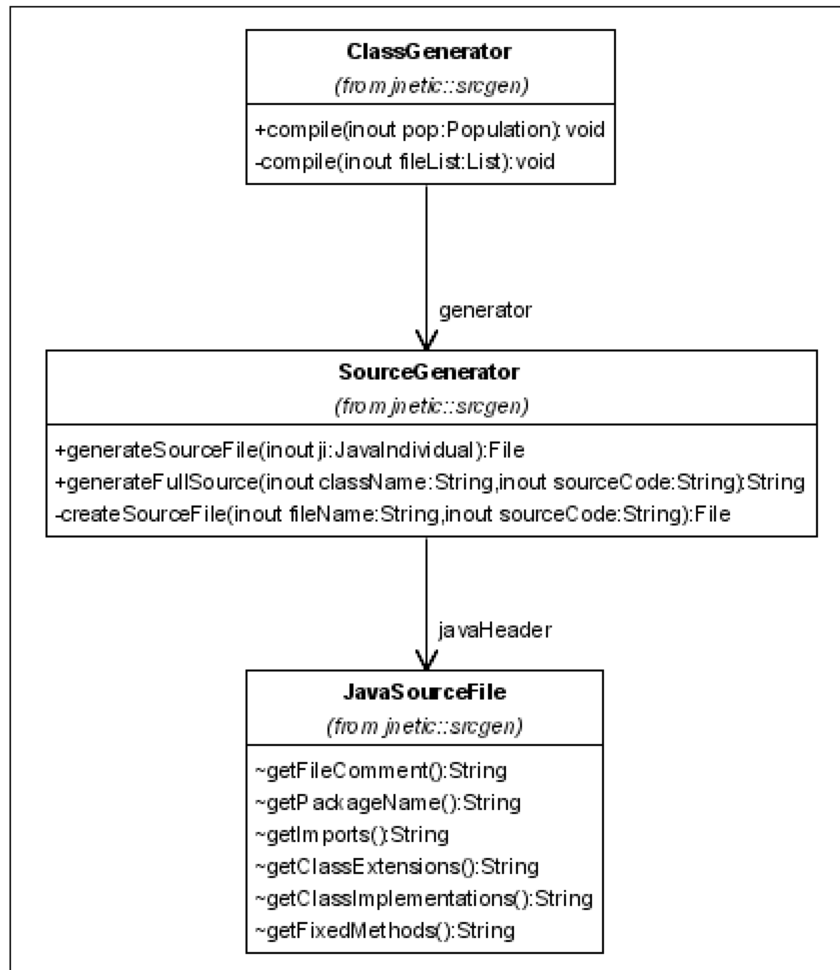


FIGURA 4.4 – Fragmento do diagrama de classes UML centrado na classe ClassGenerator

A fig. 4.5 contém um fluxograma do funcionamento do protótipo. Como pode ser observado a primeira etapa é a leitura de um arquivo de configuração XML contendo os parâmetros de execução. Então a população inicial é gerada aleatoriamente. As próximas etapas são iterativas e se repetem a cada geração. Os arquivos fontes dos indivíduos são gerados, compilados e submetidos à avaliação pela classe do usuário, se a condição de término foi atingida o programa é encerrado, senão são realizadas operações genéticas a próxima geração é iniciada.

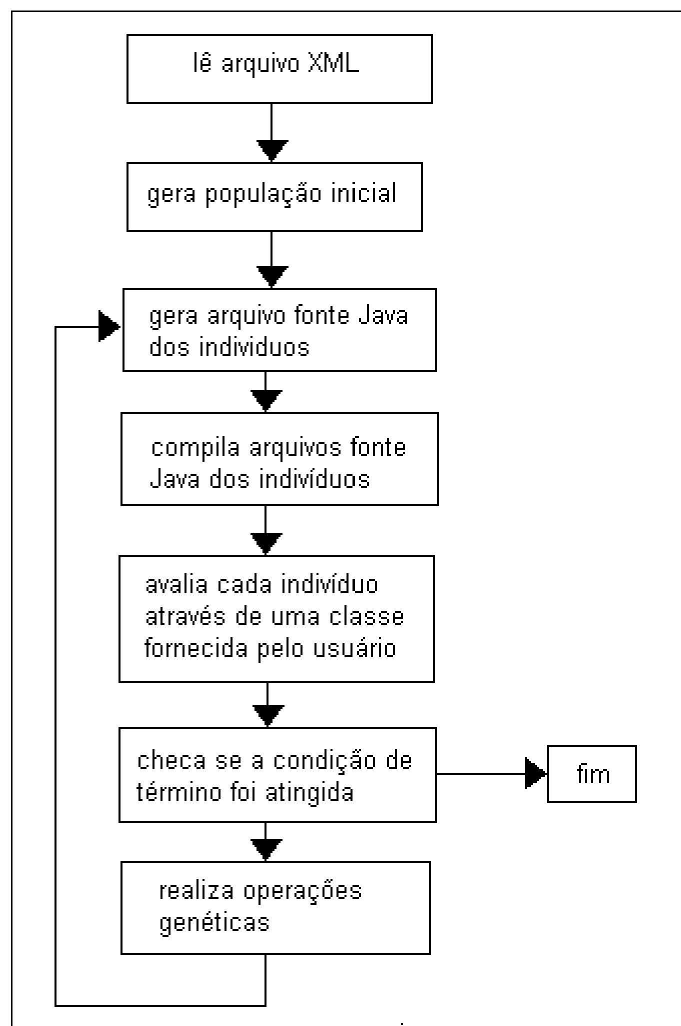


FIGURA 4.5 – Funcionamento do protótipo

4.3 IMPLEMENTAÇÃO

A ferramenta foi implementada utilizando-se a linguagem de programação Java no ambiente integrado de desenvolvimento Eclipse (ECLIPSE, 2003). As características da

ferramenta foram definidas, primeiramente pelas características da própria programação genética, e segundo, pelas necessidades da adequação as características do simulador Robocode.

4.3.1 FUNÇOES E TERMINAIS

Como o objetivo final da ferramenta era a sua utilização com o simulador Robocode isso criou a necessidades específicas para a ferramenta:

a) os tanques no simulador são programas na linguagem de programação Java. Entre as diversas possíveis abordagens para essa característica a escolhida foi em utilizar funções e terminais da própria linguagem de programação Java, dessa forma a árvore dos indivíduos da programação genética é árvore de um programa Java e podem ser diretamente executados pelo simulador Robocode. Para facilitar a implementação dessa característica à ferramenta foi arquitetada para somente utilizar as funções e terminais da linguagem de programação Java.

b) Os eventos do simulador são implementados através de vários métodos, essa característica cria a necessidade de que cada indivíduo na programação genética possua múltiplas árvores, uma para cada método. A ferramenta foi implementada com essa característica, cada indivíduo é capaz de possuir múltiplas árvores de programas.

c) Como muitos dos eventos do simulador possuem parâmetros que fornecem informações importantes sobre a situação da batalha é necessário ter acesso a tais parâmetros para satisfazer a propriedade de suficiência das funções e terminais da programação genética. Essa situação torna necessário que cada método possua um subconjunto diferente de funções e terminais que representem as informações de tais parâmetros.

Como a ferramenta utiliza somente a linguagem de programação Java como linguagem das funções e terminais, foi criada uma interface chamada “Node” para representa um tipo básico de nó, a fig. 4.6 exibe seu código fonte.

```java
/* Created on 09/10/2003 by Andre */
package jnetic.tree;

import jnetic.engine.JavaContext;
import jnetic.population.IGeneticDescriptor;
import jnetic.tree.java.Conversor;
import jnetic.tree.java.Scanner;

public interface Node {

    Class getType();
    String eval();
    void populateParams(JavaContext context,
                        IGeneticDescriptor espec, int deep);
    int getDeep(int deep);
    boolean scanSubNodes(Scanner scanner);
    Node convertSubNodes(Conversor conversor);
    Node newNode();
    Node cloneTree();
}
```

FIGURA 4.6 – Código fonte da *interface* Node

A partir dessa *interface* dois conjuntos de classes foram implementados para representar as construções da linguagem:

a) Elementos fixos da linguagem:

- BinaryOperation: representa elementos binários fixos, formados sempre por dois elementos ligados através de um símbolo da linguagem:
    - operadores matemáticos, retornam um valor numérico:
        - soma: “+”;
        - subtração: “-“;
        - multiplicação: “*”;
        - divisão: “/”;
    - operadores boleanos, retornam um valor boleano:
        - maior que: “>”;
        - menor que: “<”;
        - igual: “==”;
        - diferente: “!=”;
- JavaFluxControl: representa elementos fixos de controle de fluxo da linguagem, não possuem retorno, tomam como parâmetro qualquer nó com tipo de retorno boleano e possuem uma expressão do tipo “bloco” vinculada, são eles:
    - elemento de repetição:

- enquanto: “while”
- elemento condicional:
  - se: “if”
- Block: elemento fixo representa um bloco de código, pode conter um numero de nós limitado por um parâmetro de configuração, foi convencionado que somente nós com tipo de retorno nulo são aceitos.

b) Elementos dinâmicos:

- Constant: elemento dinâmico, o usuário da ferramenta pode definir diversos terminais do tipo Constant, o tipo de retorno depende do definido pelo usuário, representa uma constante na linguagem Java.
- Method: elemento dinâmico, representa uma chamada para uma função que não toma parâmetros, e, portanto é um terminal. Assim com elementos Constant também é definido pelo usuário e seu tipo de retorno é variável.
- Function: elemento dinâmico, representa uma chamada para uma função que recebe parâmetros. Também é definida pelo usuário, a quantidade de parâmetros e seus tipos de dados e o tipo de dado de retorno da própria Function é definido pelo usuário.

Esse conjunto de funções e terminais, devido à variedade de tipos de retorno e necessidades de tipos de dados específicos em determinados nós, não possui a propriedade *closure.* Tal fato é levado em consideração na geração da população inicial e na operação de *crossover.*

O conceito de “contexto” agrupa dois elementos:

a) declaração do método
b) conjunto de nós que somente esse método tem acesso.

Deve ser informado ao menos um contexto. Existe um contexto especial chamado “default”, os nós deste contexto estarão disponíveis para todos os outros contextos.

4.3.2 POPULAÇÃO INICIAL

A ferramenta gera a população inicial aleatoriamente. Como o conjunto de funções e terminais não possui naturalmente a propriedade *closure*, é necessário realizar essa etapa cuidadosamente. Isso é realizado através da filtragem de nós por tipo de retorno no momento

da construção da árvore. A construção de uma árvore aleatória é realizada da seguinte maneira:

a) a raiz de toda árvore (método) é um nó do tipo “Bloco”;

b) é determinado aleatoriamente a quantidade de elementos do Bloco, limitando tal numero a uma quantidade máxima fornecida pelo usuário;

c) é selecionado do conjunto de todos os nós, somente os que possuem tipo de retorno nulo, um entre estes é selecionado aleatoriamente e inserido no “Bloco”;

- se esse nó for um terminal, continua-se o povoamento do bloco.
- se for uma função povoa-se seus parâmetros da mesma maneira, através da seleção de nós compatíveis, se alguns desses nós for, também, uma função, então este também é povoado e assim por diante, até o momento que todos os pontos finais da sub-árvore sejam terminais.
- se a profundidade máxima foi atingida então se seleciona somente terminais e não todos os nós.

4.3.3 REPRODUÇÃO

A operação de reprodução é extremamente simples não apresentou qualquer dificuldade ou necessidade especial.

4.3.4 CROSSOVER

Como o conjunto de funções e terminais utilizado não possui, por si só, a propriedade de *closure*, é necessário, na operação de *crossover,* tomar cuidado para que as árvores resultantes sejam válidas. Isso fui realizado da seguinte forma:

a) Dois indivíduos são selecionados da população;

b) Se os indivíduos possuírem múltiplas árvores é determinado em qual delas será realizado o *crossover*. É fundamental que seja utilizada a árvore do mesmo método de ambos os indivíduos devido a conflitos de “contexto” (parâmetros) dos métodos;

c) É selecionado um ponto de *crossover* no árvore do individuo A;

d) Procura-se na árvore de B por todos os pontos compatíveis com o nó escolhido da árvore de A.

- Se a árvore do individuo B não possuir nenhum ponto compatível, então ambos indivíduos são inseridos inalterados.

e) É selecionado aleatoriamente um dos pontos compatíveis de B;
f) Checa-se o tamanho das árvores resultantes.
   - Se o tamanho da árvore de B ficar maior que o máximo então B é inserido inalterado, o mesmo ocorra com A.

Com essa seqüência de operações é garantido que as árvores geradas serão sempre válidas.

4.3.5 DESCRIÇÃO DO ARQUIVO FONTE JAVA

Como a ferramenta utiliza a linguagem de programação Java, é necessário gerar os arquivos fontes e compilá-los para que possam ser avaliados.

Para a geração dos arquivos fontes é necessário ter conhecimento sobre certas características dos arquivos fonte, as seguintes informações devem ser conhecidas:

a) nome do "pacote" que contém a classe;
b) os *imports* que a classe utiliza;
c) o nome das classes que a classe deriva;
d) os nomes das *interfaces* que a classe implementa;
e) os métodos fixos que a classe deve possuir;
f) opcionalmente, um comentário para ser inserido no inicio do arquivo.

4.3.6 AVALIAÇÃO DA FITNESS

A avaliação de um individuo é completamente dependente do problema dado. A ferramenta disponibiliza uma classe abstrata chamada "Evaluator" que deve ser herdada por uma classe do usuário e dada como parâmetro de execução. Através dessa classe a ferramenta poderá obter o resultado da avaliação do individuo. A fig. 4.7 contém o código fonte da classe, os comentários foram removidos.

O principal método que o usuário deve implementar é o método "eval", o retorno desse método indica se a condição de término foi atingida. A ferramenta espera que após o retorno desse método todos os indivíduos tenham seu *standarized fitness* preenchido.

Outro método que deve ser implementado é o método “close”, tal método é utilizado no final da execução do “Jnetic” para dar ao “Evaluator” a oportunidade de encerrar qualquer recurso aberto, como por exemplo, arquivos ou soquetes.

```
/* Created on 19/10/2003 by Andre */
package jnetic.engine;

import jnetic.population.Population;

/**
 * @author Andre 19/10/2003
 *
 */
public abstract class Evaluator {

    protected final boolean evalPopulation(Population pop,
                                           EventListener listener) {
        listener.startEvaluation();

        //avaliar individuos
        boolean ret = eval(pop, listener);

        //ordenar pelo fitness
        pop.sortByNormalizedFit();

        listener.endEvaluation();
        return ret;
    }

    public abstract void close();

    public abstract boolean eval(Population pop,
                                 EventListener eventRaiser);

}
```

FIGURA 4.7 – Código fonte da classe Evaluator

4.3.7 PARAMETROS GENÉTICOS

A ferramenta “Jnetic” utiliza um grande número de parâmetros para sua execução, são eles:

a) profundidade máxima da árvore na população inicial;
b) profundidade máxima da árvore durante a execução;
c) quantidade máxima de elementos em um bloco;
d) quantidade de indivíduos na população;
e) probabilidade de *crossover;*
f) probabilidade de reprodução.

g) informações sobre o arquivo Java, cap. 4.3.5;
h) lista de “contextos”, cap.

Devido a grande quantidade de elementos, essas informações devem estar contidas em um arquivo com o formato *Extensible Markup Language*, o formato do arquivo deve ser como o mostrado na fig. 4.8.

FIGURA 4.8 – Formato do arquivo XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<jnetic>
        <workDir></workDir>
        <genetic_parameters>
                <maxTreeDeep> </maxTreeDeep>
                <maxTreeDeepOnStartingPop> </maxTreeDeepOnStartingPop>
                <maxBlockHeight> </maxBlockHeight>
                <popSize> </popSize>
                <crossOverProbability> </crossOverProbability>
                <evaluator> </evaluator>
        </genetic_parameters>
        <java>
                <language>
                        <element> </element>
                </language>
                <file>
                        <package> </package>
                        <import> </import>
                        <implements> </implements>
                        <class_prefix> </class_prefix>
                        <extends> </extends>
                        <comment></comment>
                </file>
        </java>
        <contexts>
                <context>
                        <declaration> </declaration>
                        <constant>
                                <id> </id>
                                <type> </type>
                                <value> </value>
                        </constant>
                        <method>
                                <id> </id>
                                <type> </type>
                        </method>
                        <function>
                                <id> </id>
                                <type> </type>
                                <param>
                                        <type> </type>
                                </param>
                        </function>
                </context>

        </contexts>
</jnetic>
```

As características de linguagem XML não são detalhadas por serem de conhecimento comum, informações sobre XML podem ser obtidas em Kirk (2000), McLaughlin (2001) e Marchal (2000). A estrutura do arquivo possui como raiz um elemento chamado “jnetic”, esse elemento possui quatro partes:

i) “workDir”, diretório de trabalho, indica o local onde o protótipo deve criar os arquivos fonte e arquivos compilados;

j) “genetic_parameters”, agrupa os parâmetros da programação genética:

- “maxTreeDeep”, tamanho máximo das árvores de programa durante a execução, valor numérico inteiro;
- “maxTreeDeepOnStartingPop”, tamanho máximo das árvores de programa na população inicial, valor numérico inteiro;
- “maxBlockHeight”, quantidade máxima de nós em um bloco de programação Java, valor numérico inteiro;
- “popSize”: tamanho da população, valor numérico inteiro;
- “crossoverProbability”, probabilidade da operação de *crossover*, esse valor somado ao do elemento “reproductionProbability” deve ter um valor máximo é 1.0, valor numérico real.
- “reproductionProbability”, probabilidade da operação de reprodução, esse valor somado ao do elemento “crossoverProbability” deve ter um valor máximo é 1.0.
- “evaluator”, nome da classe avaliador, o caminho para o arquivo “class” da classe deve estar presente no “classpath” do protótipo, para mesma poder ser encontrada, valor numérico real.

k) “java”, agrupa elementos específicos sobre a geração de código da linguagem Java:

- “language”, agrupa elementos sobre os elementos da linguagem a serem usados como funções e terminais:
  - “element”: indica o elemento de linguagem a ser utilizado indicado, valor alfanumérico, esse elemento pode se repetir várias vezes, uma vez para cada elemento. Atualmente são suportados os seguintes elementos: “if”, “while”, “>”, “<”, “<=“, “>=”. “==”, “!=”;
- “file”, elementos que descrevem o arquivo Java:
  - “package”, nome do *package* da classe, valor alfanumérico;

- "imports", elementos que devem estar presentes na clausula *imports* do arquivo, valor alfanumérico, pode ser repetir diversas vezes.
- "implements", *interfaces*, que a classe Java deve implementar, alfanumérico;
- "class_prefix", prefixo para o nome das classes, durante a execução o protótipo irá gerar as classes com utilizando esse prefixo seguido de um numero de seqüencial, valor alfanumérico;
- "extends", indica que o nome da classe que deve estar na clausula *extends* do arquivo fonte Java.
- "comments", comentário que deve ser adicionado no arquivo fonte Java, alfanumérico;

l) "contexts", lista dos contextos (métodos) Java,
- "context", indica um contexto, podem existir vários destes elementos;
  - "declaration", declaração do contexto, é a declaração do método que o contexto representa;
  - "constant", indica um terminal do tipo constante,
    - "id", o nome da constante;
    - "type", o tipo de dado da constante;
    - "value", o valor da constante;
  - "method", indica um terminal do tipo método,
    - "id", nome do método;
    - "type", o tipo de dado que o método retorna;
  - "function", indica uma função,
    - "id", nome da função;
    - "type", tipo de dado que a função retorna;
    - "param", indica um parâmetros para a função,
      - "type", tipo de dado que o parâmetro utiliza;

4.3.8 UTILIZAÇÃO

Como a ferramenta Jnetic foi desenvolvida utilizando-se a linguagem de programação Java é necessário, para sua execução, que exista instalado no computador a ser usado o *Java Runtime Enviroment* (JRE) versão 1.4 ou superior.

A ferramenta é iniciada a partir de um console de comando (prompt de comandos no Windows ou o Shell no Linux), e recebe com argumento o nome de um arquivo XML de configuração.

A ferramenta continua em execução até que a condição de término seja atingida, tal condição é determinada pela classe avaliadora fornecida pelo usuário.

Em tal evento a ferramenta exibe na tela a localização e nome do arquivo fonte que representa o melhor individuo encontrado durante a execução.

5 ESTUDO DE CASO

Este capitulo descreve um estudo de caso da ferramenta. O objetivo foi validar o modelo especificado no capitulo anterior através da geração de tanques que duelassem com o tanque “SittingDuck”.

5.1 FUNÇÕES, TERMINAIS E EVENTOS UTILIZADOS

O conjunto de funções e terminais utilizado são os mesmos que foram apresentados no cap. 3.1.6, os eventos utilizados foram somente o “run” e o “onScannedRobot”. Inicialmente foi criado um arquivo de configuração XML para a execução do estudo de caso um fragmento deste arquivo é exibido nas fig. 5.1 e 5.2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<jnetic>
        <workDir>c:\robotDepot</workDir>
                <genetic_parameters>
                <maxTreeDeep>12</maxTreeDeep>
                <maxTreeDeepOnStartingPop>6</maxTreeDeepOnStartingPop>
                <maxBlockHeight>8</maxBlockHeight>
                <popSize>400</popSize>
                <crossOverProbability>20</crossOverProbability>
                <evaluator>RobotEvaluator</evaluator>
        </genetic_parameters>
        <java>
                <language>
                        <element>if</element>
                        <element>while</element>
                        <element>==</element>
                        <element>!=</element>
                        <element>&lt;</element>
                        <elementr>&gt;</element>
                </language>
                <file>
                        <package>jnetic</package>
                        <import>robocode.Robot</import>
                        <import>robocode.ScannedRobotEvent</import>
                        <class_prefix>JeneticRobot</class_prefix>
                        <extends>Robot</extends>
                        <comment>This is a robot</comment>
                </file>
        </java>
        <contexts>
                <context>
                        <declaration>global</declaration>
                        <constant>
                                <id>DEGREES_15</id>
                                <type>Double</type>
                                <value>15.0</value>
                        </constant>
                        <constant>
                                <id>DEGREES_30</id>
                                <type>Double</type>
                                <value>30.0</value>
                        </constant>
                        <constant>
```

FIGURA 5.1 – Primeira parte do arquivo XML utilizado com o Robocode.

```
<constant>
        <id>DEGREES_45</id>
        <type>Double</type>
        <value>45.0</value>
</constant>
<constant>
        <id>DEGREES_60</id>
        <type>Double</type>
        <value>60.0</value>
</constant>
<constant>
        <id>DEGREES_90</id>
        <type>Double</type>
        <value>90.0</value>
</constant>
<constant>
        <id>DEGREES_180</id>
        <type>Double</type>
        <value>180.0</value>
</constant>

<method>
        <id>getBattleFieldHeight()</id>
        <type>Double</type>
</method>
<method>
        <id>getHeading()</id>
        <type>Double</type>
</method>
<method>
        <id>getX()</id>
        <type>Double</type>
</method>
<method>
        <id>getY()</id>
        <type>Double</type>
</method>
<method>
        <id>fire(1)</id>
        <type>null</type>
</method>
<method>
        <id>fire(2)</id>
        <type>null</type>
</method>
<method>
        <id>fire(3)</id>
        <type>null</type>
</method>
<method>
        <id>scan()</id>
        <type>Double</type>
</method>
```

FIGURA 5.2 – Segunda parte do arquivo XML utilizado com o Robocode.

5.2 CLASSE AVALIADORA

Para a avaliação dos indivíduos foi implementada a classe “RobotEvaluator” herdeira da classe “Evaluator” oferecida pelo protótipo. Essa classe instancia um objeto da classe “RobocodeManager” oferecida pelo simulador Robocode e utilizada para comunicação com o mesmo, o código fonte dessa classe é exibido na fig. 5.3.

```
public boolean eval(Population pop, EventListener eventRaiser) {
    RobotSpecification[] rEspec = manager.getLocalRepository();
    boolean victory = false;

    //obter os componentes de todas as batalhas
    for (int i = 0; i < pop.pop.length; i++) {
        //determinar quais individuos devem ter o fitness calculado
        if (pop.pop[i].needEvaluation()) {
            for (int j = 0; j < rEspec.length; j++) {
                if (rEspec[j].getClassName().equals(pop.pop[i].getFullClassId())) {
                    //executar batalha
                    RobotSpecification[] rs = new RobotSpecification[2];
                    rs[0] = theEnemy;
                    rs[1] = rEspec[j];
                    BattleSpecification espec = new BattleSpecification(
                                            runsPerBattle, bfEspec, rs);
                    manager.runBattle(espec);
                    synchronized (this) {
                        try {
                            wait();
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    if (elementRawFit > theEnemyRawFit)
                        victory = true;
                    double raw = (double) elementRawFit / (double) runsPerBattle;
                    pop.pop[i].rawFitness = raw;
                    pop.pop[i].standarizedFit = 290 - (raw / 3.0);
                }
            }
        }
    }

    if (currentGeneration == maxGeneration) {
        return true;
    } else {
        currentGeneration++;
        return false;
    }

}
```

FIGURA 5.3 – Fragmento do código fonte da classe avaliadora dos tanques.

A avaliação de cada indivíduo foi realizada através de batalhas do tipo duelo, onde cada individuo testado realizou três batalhas contra o tanque "SittingDuck" que acompanha o simulador Robocode.

O *raw fitness* dos indivíduos foi calculado através da média da pontuação atingida nas três batalhas.

Como no simulador Robocode uma grande pontuação indica um bom tanque é necessário ajustar o *raw fitness* para se obter um valor de *standarized fitness* correto.

O *standarized fitness* foi calculado através da subtração do valor do *raw fitness* da constante 290. O autor não foi capaz de determinar a pontuação teórica máxima de um tanque no robocode, portanto diversos testes empíricos foram realizados para determinar tal valor. Nesses testes a pontuação máxima atingida foi de "290,0". Essa pontuação foi obtida através de um tanque que, sem efetuar disparos, destruiu outros tanques somente através de colisões, dessa forma ganhando uma pontuação extra, conforme apresentado no cap. 3.1.3.

5.3 CONDIÇÃO DE TÉRMINO

Foi escolhido como condição de término quando qualquer indivíduo obtivesse uma pontuação média maior que o "SittingDuck" ou a 50 ª geração fosse atingida.

5.4 RESULTADOS

A avaliação dos indivíduos demonstrou ser extremamente lenta, todo o processo de geração de fonte, compilação, avaliação e operações genéticas de cada geração leva aproximadamente 912 segundos, pouco mais de 15 minutos, a fig. 5.4 exibe a divisão de tempo, em porcentagem aproximada em cada uma dessas etapas.

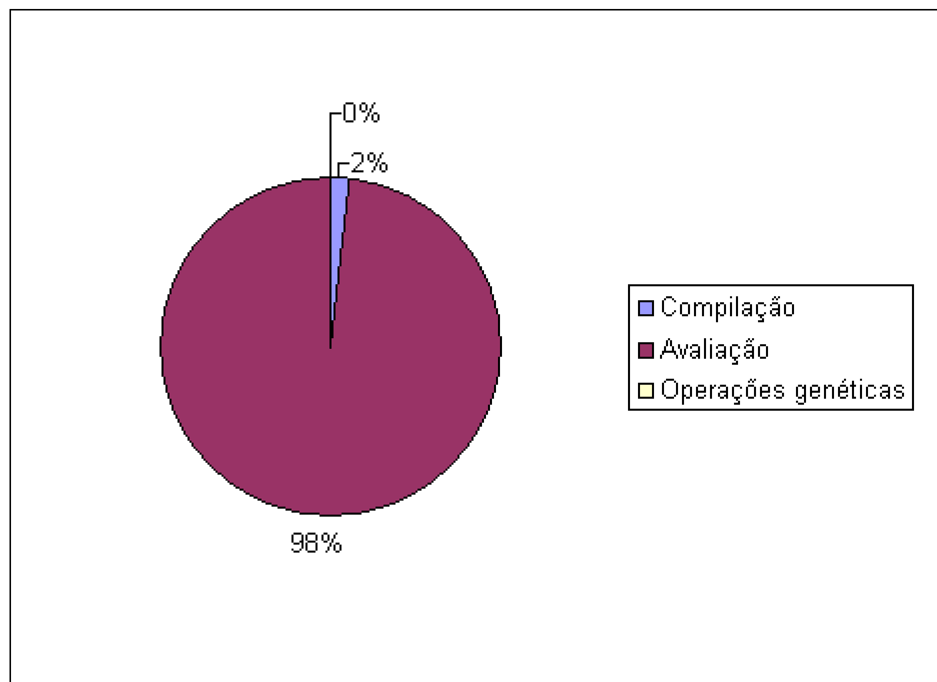


FIGURA 5.4 – Divisão aproximada do tempo de cada geração.

Fica evidente que o grande gargalo da execução é a avaliação dos indivíduos através do simulador Robocode.

No estudo de caso, somente na 12ª geração foi encontrar um individuo que capaz de derrotar o “SittingDuck” todas as vezes que entra em batalha com o mesmo. O tempo total decorrido até tal programa ser encontrado foi de 10032 segundos, ou pouco mais de 2 horas e 45 minutos.

A fig. 5.5 exibe um gráfico com a evolução da média da *adjusted fitness* em toda população em cada geração.Demonstrando um crescente aumento da mesma a cada geração.

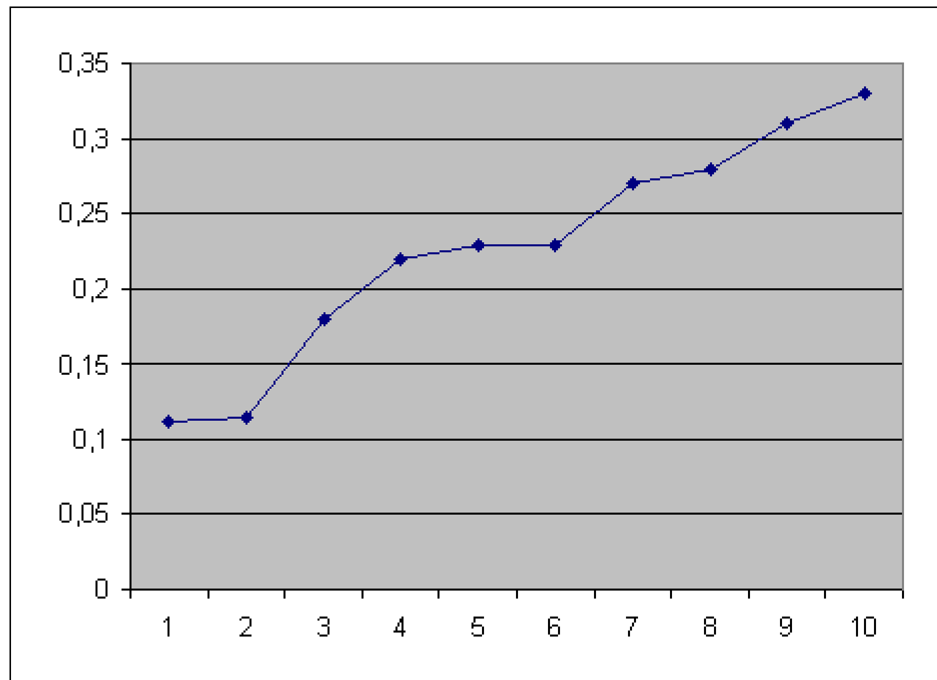


FIGURA 5.5 – Histórico de evolução média da *adjusted fitness*

A fig. 5.6 exibe um indivíduo da população inicial. Esse individuo foi gerado aleatoriamente pela ferramenta e não foi capaz de obter nenhum ponto no simulador Robocode.

```
/*Generated by Jnetic*/
package jnetic;
import robocode.Robot;
import robocode.ScannedRobotEvent;
public class j291 extends Robot {
    public void run() {
        turnLeft(getX());
        scan();
    }
    public void onScannedRobot(ScannedRobotEvent event) {
        turnGunLeft(getBattleFieldWidth());
        fire(1);
        turnLeft(getY());
        turnLeft(getHeading());
    }
}
```

FIGURA 5.6 – Indivíduo da população inicial.

A fig. 5.7 exibe o individuo encontrado. Esse individuo é capaz de derrotar, por pontos, o tanque “SittingDuck” sempre que entram em batalha. O código deste programa é visivelmente mais complexo do que o do programa da geração inicial e é de difícil compreensão, ou seja, não é possível identificar exatamente o que o programa está fazendo.

```
/*Generated by Jnetic*/
package jnetic;
import robocode.Robot;
import robocode.ScannedRobotEvent;
public class j2901 extends Robot {
    public void run() {
        while ((getHeading() > getBattleFieldWidth())) {
            scan();
            while ((getBattleFieldWidth() != getY())) {
                turnGunRight(getY());
            };
            fire(2);
            scan();
        };
        ahead(getBattleFieldWidth());
        turnRight(getY());
        turnGunRight(getX());
        turnGunRight(getX());
    }
    public void onScannedRobot(ScannedRobotEvent event) {
        fire(3);
        fire(1);
        if ((getBattleFieldWidth() != getBattleFieldHeight())) {
            ahead(getBattleFieldWidth());
            fire(2);
            scan();
            fire(2);
        };
    }
}
```

FIGURA 5.7 – Melhor indivíduo encontrado.

6 CONCLUSÕES

O protótipo desenvolvido obteve êxito em sua tarefa, pois realiza com sucesso operações de programação genética. Os problemas residem na própria aplicação da programação genética no simulador Robocode. Foi observado que um bom tanque no Robocode possui, geralmente, um código fonte pequeno, simples, eficiente e rico em construções avançadas de programação e estruturas de dados, ou seja, são programas especializados e minimalistas em suas operações. A maioria dos tanques gerados pelo protótipo, mesmo os melhores, executavam diversas operações "inúteis", e, devido ao conjunto de nós utilizados, não eram capazes de nenhuma construção complexa de programação. Possíveis soluções para esse problema poderiam ser uma ampliação do conjunto de funções e terminais, incluindo, por exemplo, variáveis e objetos temporários, diminuição da profundidade da árvore e talvez um aumento do tamanho da população.

Foi observada também uma convergência prematura da população, isso se deve ao método utilizado para a medição da *fitness*. Com tal método 90% dos indivíduos obtém pontuação zero no Robocode, e a nova geração passa a ser composta pelos 10% restantes, eliminando assim a variedade dos programas. Para solucionar essa convergência é necessário encontrar outra forma para avaliar os indivíduos ou utilizar operações genéticas secundárias, tais como "mutação".

Outro grande problema, talvez o principal encontrado, é tempo de avaliação dos indivíduos pelo simulador Robocode. Aparentemente não há solução simples, se é que existe solução, para esse problema.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, S. W. **Análise e projeto orientados a objeto** : um guia para o desenvolvimento de aplicações orientadas a objeto. 4. ed. Rio de Janeiro: Infobook, 1997.

BOOCH, G. **UML – guia do usuário.** 5. ed. Rio de Janeiro: Campus, 2000.

ECLIPSE. Version 3.0, [S.l : s.n], 2003. Disponível em: <http://www.eclipse.org>. Acesso em: 3 set. 2003.

FLANAGAN**,** David. **Java** : o guia essencial**.** Rio de Janeiro: Campus, 2000.

GENTLEWARE. **PoseidonUML.** Hamburg : [s.n], 2003 . Disponível em: <http://www.gentleware.com>. Acesso em: 3 set. 2003.

HAGGAR, P. **Java – guia prático de programação.** Rio de Janeiro: Campus, 2000.

IBM**. Robocode,** [S.l], jul. 2001. Disponível em: <http://www.alphaworks.ibm.com/tech/robocode>. Acesso em: 1 set. 2003.

KIRK, C. **XML black book.** São Paulo: Makron Books, 2000.

KOZA, J. R. **Genetic programing** : on the programming of computers by means of natural selection. 4. ed. Pittsfield: The MIT Press, 1992.

Li S. **Rock 'em, sock 'em robocode!** [S.l]: Wrox Press, jan. 2002. Disponível em: <http://stsmaxi.infoseek.livedoor.net/>. Acesso em: 10 nov. 2003.

MARCHAL, B. **XML conceitos e aplicações.** São Paulo: Berkeley, 200.

MCLAUGHLIN, B. **Java & XML.** 2. ed. Sebastopol: O'Reilly, 2001.

NEWMAN, A. **Usando java.** Rio de Janeiro: Campus, 1997.