Fábio Duncan de Souza

TOGAI - UMA FERRAMENTA PARA IMPLEMENTAÇÃO DE ALGORITMOS GENÉTICOS

Campos dos Goytacazes

Março de 2008

UNIVERSIDADE CANDIDO MENDES - UCAM
PROGRAMA DE PÓS-GRADUAÇÃO EM PESQUISA OPERACIONAL
E INTELIGÊNCIA COMPUTACIONAL

Fábio Duncan de Souza

TOGAI - UMA FERRAMENTA PARA IMPLEMENTAÇÃO DE ALGORITMOS GENÉTICOS

Dissertação apresentada à Universidade Candido Mendes -UCAM-Campos, como parte dos requisitos para obtenção do túlo de Mestre em Pesquisa Operacional e Inteligência Computacional.

Orientador: Prof. DSc. Dalessandro Soares Vianna

Campos dos Goytacazes

2008

FÁBIO DUNCAN DE SOUZA

TOGAI - UMA FERRAMENTA PARA IMPLEMENTAÇÃO DE ALGORITMOS GENÉTICOS

Dissertação apresentada à Universidade Candido Mendes -UCAM-Campos, como parte dos requisitos para obtenção do túlo de Mestre em Pesquisa Operacional e Inteligência Computacional.

BANCA EXAMINADORA

Prof. DSc. Dalessandro Soares Vianna - Orientador
(Universidade Candido Mendes - Campos/RJ)

Prof. DSc. Edwin Benito Mitacc Meza - Banca
(Universidade Candido Mendes - Campos/RJ)

Prof[a]. DSc. Jacqueline Magalhães Rangel Cortes - Banca
(Universidade Estadual do Norte Fluminense Darcy Ribeiro - Campos/RJ)

Prof. DSc. Marcone Jamilson Freitas Souza - Banca
(Universidade Federal de Ouro Preto - Ouro Preto/MG)

Campos dos Goytacazes - RJ
2008

A minha mãe

Neusa Duncan de Souza

e a minha esposa e filhos

Mônica Viana Ribeiro Gomes,

Maria Clara Ribeiro Gomes Duncan de Souza,

Miguel Ribeiro Gomes Duncan de Souza,

com amor…

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus por permitir que caminhos fossem trilhados até a chegada deste momento. Agradeço também, em especial, ao

Doutor Dalessandro pelas orientações, pela paciência e principalmente pela amizade que me ofereceu ao longo deste trabalho.

Agradeço aos amigos que compartilharam e ajudaram a superar os desafios deste mestrado.

Por fim, agradeço às seguintes instituições pelo apoio financeiro: Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ), Parque de Alta Tecnologia do Norte Fluminense (TECNORTE) e Fundação Estadual do Norte Fluminense (FENORTE).

Resumo

TOGAI - UMA FERRAMENTA PARA IMPLEMENTAÇÃO DE ALGORITMOS GENÉTICOS

Problemas de otimização combinatória aparecem freqüentemente em vários setores da economia. Grande parte deles são intratáveis (não possuem um algoritmo polinomial para sua solução) por natureza ou são grandes o suficiente para tornar inviável o uso de algoritmos exatos. O uso de Algoritmos Genéticos (AGs) é uma das principais estratégias para resolver este tipo de problema. Um AG é formado de várias etapas. Algumas delas (a etapa de seleção, por exemplo) possuem diversas políticas diferentes que podem ser reaproveitadas de um problema para outro, desde que a implementação do AG esteja bem modelada. Este trabalho propõe uma ferramenta de auxílio à implementação que diminui o trabalho do usuário ao desenvolver um algoritmo genético para um determinado problema. Além disso, a ferramenta torna a implementação do AG organizada (modulada) o suficiente para evitar o "retrabalho" com a modificação de certos métodos. Por exemplo, uma vez implementado um AG, é possível testar diferentes políticas de seleção de indivíduos (já implementadas e disponíveis pela ferramenta) neste AG sem a necessidade de novas codificações.

PALAVRAS-CHAVE: Algoritmos Genéticos, Modelo de Implementação, Reuso de Código.

Abstract

TOGAI - TOOL FOR GENETIC ALGORITHMS IMPLEMENTATION

Combinatorial optimization problems are common on several economy sectors. Most of them are untreatable (it is not known a polynomial algorithm to solve them) or are large enough to make the use of exact algorithms a bad strategy. The use of Genetic Algorithms (GAs) is one of the main strategies to solve this kind of problem. A GA is formed by many phases. Some of them (the selection phase, for instance) have different politics and each of them can be reused from a problem to another, since the GA implementation is well modeled. This work proposes a tool to suport the implementation that reduces the user work on developing a genetic algorithm for a problem. Besides, the tool makes the GA implementation organized enough to avoid the "rework" with some modifications. For instance, once we have an implemented GA, it is possible to test different selection of individual politics (which are already implemented and available by the tool) in this GA without new codifications.

KEYWORDS: Genetic Algorithms, Implementation Model, Code Reuse.

Sumário

Lista de Figuras

1 Introdução **12**

2 Algoritmos Genéticos **15**

2.1 Analogia e História 15

2.2 Aplicação 16

2.3 O Algoritmo Genético Simples 17

2.4 Representação do Cromossomo 19

2.5 Avaliação do Indivíduo 21

2.6 População Inicial 21

2.7 Seleção de Indivíduos 22

2.7.1 Roleta (Seleção Estocástica com Reposição - Stochastic Sampling with Replacement) 22

2.7.2 Seleção Aleatória (Stochastic Universal Sampling - SUS) 23

2.7.3 Torneio (Variação Aleatória) 24

2.7.4 Torneio (Variação Roleta) 24

2.7.5 Seleção Determinística (Deterministic Sampling - DS) 24

2.7.6 Seleção Estocástica por Resto Sem Reposição (Stochastic Remainder Sampling) 25

2.7.7 Seleção Estocástica por Resto Com Reposição (Stochastic Remainder Sampling with Replacement) 25

2.7.8 Seleção Estocástica Sem Reposição (Stochastic Sampling without Replacement) 26

2.7.9 Seleção por *ranking* 26

2.8 Cruzamento . . . 27
2.8.1 Cruzamento com um Ponto de Corte . . . 27
2.8.2 Cruzamento com *N* Pontos de Corte . . . 27
2.8.3 Cruzamento Uniforme . . . 28
2.8.4 Cruzamento Baseado em Ordem . . . 29
2.9 Mutação . . . 29
2.10 Evolução da População . . . 30
2.10.1 Simple Genetic Algorithm (SGA) . . . 30
2.10.2 Steady State Genetic Algorithm (SSGA) . . . 30
2.10.3 Parâmetro Gap da Geração . . . 31
2.11 Critério de Parada . . . 31
2.11.1 Número Máximo de Gerações . . . 31
2.11.2 Melhora Mínima da Solução em um Determinado Número de Gerações 31
2.11.3 Convergência da População . . . 32
2.11.4 Tempo de Computação . . . 32
2.11.5 Valor Alvo . . . 32
2.12 Parâmetros dos Algoritmos Genéticos . . . 32
2.12.1 Tamanho da População (*SIZE_POP*) . . . 33
2.12.2 Probabilidade de Cruzamento (*pc*) . . . 33
2.12.3 Probabilidade de Mutação (*pm*) . . . 33
3 Trabalhos Relacionados 34
4 A Ferramenta TOGAI (Tool for Genetic Algorithms Implementation) 36
4.1 O modelo de Desenvolvimento dos Algoritmos Genéticos . . . 37
4.1.1 População Inicial . . . 39
4.1.2 Avaliação do Indivíduo . . . 39
4.1.3 Seleção de Indivíduos . . . 40

4.1.4 Cruzamento 42
4.1.5 Mutação 42
4.1.6 Evolução da População 43
4.1.7 Critério de Parada 46
4.2 Detalhes de Implementação do Modelo 47
4.2.1 Constantes 47
4.2.2 Funções Complementares 47
4.3 Utilização da Ferramenta TOGAI 51

5 Casos de Uso **56**

5.1 Problema da Mochila 57
5.1.1 Implementação do Problema da Mochila Binária utilizando Algoritmos Genéticos 58
5.1.2 Testes Computacionais 60
5.1.3 Conclusão do Caso de Uso 64
5.2 Problema da Filogenia 64
5.2.1 Implementação do Problema da Filogenia utilizando AGs 67
5.2.2 Testes Computacionais 71
5.2.3 Conclusão do Caso de Uso 75

6 Trabalhos Futuros **76**

7 Conclusão **78**

Referências Bibliográficas **80**

Apêndice A – Código Fonte da Ferramenta TOGAI **85**

A.1 Código fonte da estrutura principal 85
A.2 Código fonte das políticas de seleção 91
A.3 Código fonte das políticas de evolução da população 107
A.4 Código fonte dos critérios de parada 113

Lista de Figuras

2.1 Fluxograma de um Algoritmo Genético Simples. 18

2.2 Cromossomo e seqüência de DNA com gene em destaque. 19

2.3 Tabela comparativa: Genética Natural X Algoritmo Genético. 20

2.4 Exemplo de uma Roleta para Seleção de Indivíduos. 23

2.5 Exemplo de Geração de uma População Auxiliar. 25

2.6 Exemplo de Seleção por *Ranking*. 26

2.7 Exemplo de Cruzamento com Um Ponto de Corte. 27

2.8 Exemplo de Cruzamento com *N* Pontos de Corte (onde *N*=2). 28

2.9 Exemplo de Cruzamento Uniforme. 28

2.10 Exemplo de Mutação. 30

4.1 Definição, na linguagem *C ANSI*, das estruturas que formam um indivíduo. . . 38

4.2 Procedimento que gera a população inicial. 39

4.3 Procedimento que seleciona indivíduo para cruzamento utilizando a política torneio (variação aleatório). 41

4.4 Algoritmo que seleciona indivíduo para cruzamento utilizando a política torneio (variação aleatório). 42

4.5 Procedimento que implementa a evolução da população SGA. 44

4.6 Algoritmo que implementa a evolução da população SGA. 45

4.7 Procedimento que atualiza a população. 45

4.8 Procedimento que implementa o critério de parada número máximo de gerações. 46

4.9 Função que desaloca a população. 48

4.10 Função que copia indivíduo. 48

4.11 Função para manter pais na nova geração. 50

4.12 Algoritmo para manter pais na nova geração. 51

4.13 Arquivos de configuração das políticas disponíveis na TOGAI. 53

4.14 Interface da ferramenta TOGAI. 54

5.1 Melhores resultados das políticas de seleção para 10 execuções com cada instância. 62

5.2 Ranking das políticas de seleção em face dos melhores resultados. 62

5.3 Gráfico comparativo do *ranking* das políticas de seleção em face dos melhores resultados. 62

5.4 Resultados médios das políticas de seleção para 10 execuções com cada instância. 63

5.5 *Ranking* das políticas de seleção em face dos resultados médios. 63

5.6 Gráfico comparativo do *ranking* das políticas de seleção em face dos resultados médios. 64

5.7 Exemplo de um conjunto de taxons e uma filogenia associada. 66

5.8 Exemplo de uma filogenia bem avaliada. 68

5.9 Exemplo de uma filogenia bem avaliada. 69

5.10 Melhores resultados para as oito instâncias da literatura. 71

5.11 Melhores resultados das políticas de seleção para 5 execuções com cada instância. 72

5.12 Ranking das políticas de seleção em face dos melhores resultados. 73

5.13 Gráfico comparativo do *ranking* das políticas de seleção em face dos melhores resultados. 73

5.14 Resultados médios das políticas de seleção para 5 execuções com cada instância. 74

5.15 Ranking das políticas de seleção em face dos resultados médios. 74

5.16 Gráfico comparativo do *ranking* das políticas de seleção em face dos resultados médios. 74

1 Introdução

Problemas de otimização combinatória aparecem freqüentemente em vários setores da economia. Grande parte dos problemas de otimização são intratáveis por natureza ou são grandes o suficiente para tornar inviável o uso de algoritmos exatos (VIANNA; OCHI; DRUMMOND, 1999). Atualmente, a principal estratégia para solucionar este tipo de problema é fazer uso de metaheurísticas. Estas são heurísticas direcionadas à otimização global de um problema, pois geram procedimentos de busca em vizinhanças que objetivam evitar a parada prematura em ótimos locais e proporcionar soluções satisfatórias.

Os Algoritmos Genéticos (AGs) são metaheurísticas pertencentes a uma classe de algoritmos de pesquisa probabilística e de otimização chamada Algoritmos Evolucionários. Estes são baseados no modelo de evolução orgânica, onde a natureza é a fonte de inspiração (DEB, 2001). Os AGs consistem na analogia entre otimização e os mecanismos da genética, combinando os conceitos de adaptação seletiva, troca de material genético e sobrevivência dos indivíduos mais capazes (GOLDBERG, 1989) (HOLLAND, 1975).

Os AGs são iniciados com um conjunto de soluções (denominadas cromossomos) chamado população. Soluções que formam uma população são utilizadas para, através de cruzamentos, formar uma nova população. Isto é motivado pela esperança de que a nova população seja melhor do que a primeira. A seleção de indivíduos da população para formar novas gerações é feita de acordo com uma certa adequação. Normalmente quanto melhores forem, mais chances de reprodução estes indivíduos terão. Este procedimento é repetido até atingir um critério de parada.

Uma característica existente nos algoritmos genéticos é o número de variações possíveis que podem ser implementadas de acordo com o problema para o qual se deseja encontrar uma solução (GOLDBERG, 1989). Porém, uma vez implementado o código de um Algoritmo Genético para um determinado problema, a reutilização deste código para um problema distinto pode ser pequena.

Este trabalho tem por objetivo a construção de uma ferramenta para implementação de AGs, que facilite e otimize o processo de desenvolvimento através da reutilização de código.

As etapas existentes nos AGs, que forem comuns a diferentes problemas, serão codificadas. Se estas etapas possibilitarem a implementação de políticas variadas, então estas serão disponibilizadas para que possam ser utilizadas pelos usuários, visando a detecção das melhores formas de solucionar seus problemas.

Para a ferramenta foi definido um modelo de implementação que combina criações e mudan-ças automáticas de código com programação e personalização do código pelo usuário. Utilizando este modelo, os usuários que possuírem problemas computacionalmente difíceis a serem solucionados, terão como fatores motivadores a facilidade e a agilidade no desenvolvimento de soluções baseadas em algoritmos genéticos. As funcionalidades propostas permitirão ao usuário fazer combinações de características e parâmetros na sua aplicação genética, muitas vezes eliminando a necessidade de nova codificação, possibilitando assim que testes de desempenho sejam facilmente realizados com o objetivo de encontrar as melhores políticas para o problema.

A ferramenta foi desenvolvida na linguagem JAVA, devido as suas características multiplataforma. Já a codificação gerada, foi implementada na linguagem C ANSI, seguindo as recomendações de codificação propostas por Staa(2000) (STAA, 2000). A linguagem C foi escolhida por ser comumente usada no meio científico e acadêmico bem como pelo seu desempenho de linguagem compilada. Inicialmente, a linguagem C++ foi selecionada para, junto de técnicas de orientação a objetos, formar um *framework* que seria utilizado pelo usuário para montar sua implementação de AG. Porém esta técnica não foi mantida, pois apesar de vantagens como economia a longo prazo e redução da manutenção, desvantagens relacionadas a maior esforço de aprendizado e dificuldades de depuração dos programas (TALIGENT, 1993) (TALIGENT, 1994) são características que vão de encontro aos objetivos deste trabalho.

A presente dissertação encontra-se organizada da seguinte forma:

Os Algoritmos Genéticos serão discutidos detalhadamente no Capítulo 2, onde são apresentadas suas aplicações, história e funcionamentos. Também são apresentadas políticas encontradas na literatura, que modificam o AG simples proposto por Holland (HOLLAND, 1975), visando ao ganho de desempenho para diferentes domínios de aplicação.

No Capítulo 3 são apresentados os trabalhos relacionados com a ferramenta TOGAI, proposta nesta dissertação.

O modelo de implementação proposto pela ferramenta TOGAI, juntamente com suas características e instruções de uso são descritos no Capítulo 4.

No Capítulo 5 são apresentados os problemas da mochila e da filogenia, utilizados como casos de uso para validar a ferramenta TOGAI. Para cada um destes problemas são apresentados

os resultados dos testes realizados, juntamente com gráficos comparativos e suas respectivas conclusões.

As conclusões finais serão apresentadas no Capítulo 6.

2 Algoritmos Genéticos

A inspiração para a criação dos Algoritmos Genéticos vem da natureza. Da mesma forma como os seres vivos evoluem através das gerações, os AGs objetivam melhorar as soluções existentes para um problema em questão. Como ocorre na natureza, nos AGs indivíduos mais aptos se sobressaem, ocorrem reproduções e mutações, nascem indivíduos e surgem novas gerações. Neste capítulo serão apresentados os funcionamentos destas fases do AG, bem como, sua história e aplicações.

2.1 Analogia e História

Os Algoritmos Genéticos se inspiram na teoria de evolução através da seleção natural proposta por Charles Robert Darwin em 1858. Darwin observou que as espécies se adaptavam ao ambiente no qual viviam. Indivíduos mais capazes de sobreviver neste ambiente eram naturalmente selecionados e tinham maior probabilidade de procriação. Conseqüentemente, eram estes quem mais passariam adiante suas características, gerando um ciclo evolutivo seletivo. Entretanto, a teoria da Evolução Natural só foi aceita muito mais tarde quando a tecnologia permitiu o conhecimento e estudo dos genes e da mutação.

Por volta de 1900, a moderna teoria da evolução combinou a genética e as idéias de Darwin sobre a seleção natural, criando o princípio básico de Genética Populacional: a variabilidade entre indivíduos em uma população de organismos que se reproduzem sexualmente é produzida pela mutação e pela recombinação genética. Os genes explicaram como os ancestrais passam suas informações aos seus descendentes no processo evolutivo e a mutação explicou o surgimento de uma característica nunca antes registrada numa população, o que implica na variabilidade da espécie. Este princípio foi desenvolvido durante os anos 30 e 40, por biólogos e matemáticos de importantes centros de pesquisa. Nos anos 50 e 60, muitos biólogos começaram a desenvolver simulações computacionais de sistemas genéticos com o objetivo de simular os processos vitais do ser humano em um computador.

Os algoritmos evolucionários, em especial os Algoritmos Genéticos, tiveram seu pri-

meiro impulso com a publicação do livro intitulado ”Adaptation in Natural and Artificial Systems”, por John Holland em 1975 (HOLLAND, 1975) (LINDEN, 2006). No entanto, somente com a proliferação dos computadores em centros de pesquisas é que a pesquisa científica nessas áreas teve seu crescimento acelerado. Nos anos 80, David E. Golderg, aluno de Holland, obteve o primeiro sucesso em aplicação industrial de Algoritmos Genéticos e publicou o livro 'Genetic Algorithms in Search, Optimization and Machine Learning' , uma das referências mais citadas na área de ciência da computação (HIRSCH, 2005).

2.2 Aplicação

Atualmente os princípios apresentados por Darwin são imitados na construção de Algoritmos Genéticos, através da evolução de populações de soluções codificadas através de cromossomos artificiais. Estes algoritmos têm recebido especial atenção nos últimos tempos por se tratarem de métodos robustos, capazes de fornecer soluções de alta qualidade para problemas considerados intratáveis por métodos tradicionais de otimização, concebidos para problemas lineares, contínuos e diferenciáveis. Mas, como é observado em (SCHWEFEL, 1994), o mundo real é não-linear e dinâmico, cheio de fenômenos como descontinuidade, instabilidade estrutural e formas geométricas fractais. Em problemas em que se precisa levar em conta tais fenômenos, os métodos tradicionais certamente não apresentarão desempenho satisfatório. Métodos evolutivos são uma alternativa para tentar superar as limitações apresentadas por métodos tradicionais, embora não garantam a obtenção da solução exata.

Michalewicz (1994) (MICHALEWICZ, 1994) cita alguns exemplos de problemas complexos de otimização como: Otimização de Funções Matemáticas, Otimização Combinatória, Otimização de Planejamento, Problema do Caixeiro Viajante, Problema de Otimização de Rota de Veículos, Otimização de *Layout* de Circuitos, Otimização em Negócios e Síntese de Circuitos Eletrônicos. Estes problemas possuem como características grandes espaços de busca, diversos parâmetros que precisam ser combinados em busca da melhor solução, e muitas restrições ou condições que não podem ser representadas matematicamente. Tais características classificam estes problemas como intratáveis.

Nos problemas intratáveis, o espaço de solução é tão amplo que pode-se considerar que o problema não tem solução através de métodos de enumeração exaustiva. Em termos práticos, um problema é tratável se o seu limite superior de complexidade é polinomial, e é intratável se o limite superior de sua complexidade é exponencial (TOSCANI; VELOSO, 2005), isto é, se seu tempo de execução é da ordem de uma função exponencial ou fatorial.

Como apresentado, utilizar métodos exatos para resolver problemas de otimização com-

binatória é inviável. Neste contexto, as Heurísticas e Metaheurísticas permitem resolver problemas com grandes instâncias, gerando boas soluções, em tempos computacionais satisfatórios (CAHON; MELAB; TALBI, 2004). Uma Heurística é uma técnica de otimização que usa algoritmos exploratórios para encontrar a solução de problemas. As soluções são buscadas por aproximações sucessivas, avaliando-se os progressos alcançados, até que o problema seja resolvido. Trata-se de métodos em que, embora a exploração seja feita de forma algorítmica, o progresso é obtido pela avaliação puramente empírica do resultado. As Metaheurísticas são heurísticas genéricas que se adaptam facilmente as estruturas paralelas e são direcionadas à otimização global de um problema, podendo conter diferentes procedimentos heurísticos de busca local na solução a cada passo. As Metaheurísticas quando aplicadas a problemas de otimização, tem como um de seus objetivos, gerar procedimentos de busca em vizinhança que evitem uma parada prematura em ótimos locais, proporcionando soluções melhores. Nas últimas décadas, surgiram vários procedimentos enquadrados como Metaheurísticas na solução de diversos problemas altamente combinatórios. Algumas das mais amplamente divulgadas são: Algoritmos Genéticos, *Simulated Annealing*, Busca Tabu, *GRASP*, Busca em Vizinhança Variável (VNS) e Colônia de Formigas.

Os Algoritmos Genéticos pertencem à classe dos algoritmos probabilísticos, mas não são métodos de busca puramente aleatórios, pois combinam elementos de métodos de busca diretos e estocásticos. Outra propriedade importante dos Algoritmos Genéticos é que eles mantêm uma população de soluções candidatas, enquanto que os métodos alternativos, como *Simulated Annealing*, processam um único ponto no espaço de busca a cada instante.

O processo de busca realizado pelos Algoritmos Genéticos é multidirecional, através da manutenção de soluções candidatas, e encoraja a troca de informação entre as direções. A cada geração, soluções relativamente "boas"se reproduzem, enquanto que soluções relativamente "ruins"são eliminadas. Para fazer a distinção entre diferentes soluções é empregada uma função de aptidão (de avaliação ou de adequação) que simula o papel da pressão exercida pelo ambiente sobre o indivíduo.

2.3 O Algoritmo Genético Simples

O algoritmo proposto por Holland (HOLLAND, 1975), é conhecido na literatura como *Simple Genetic Algorithm* ou *Standard Genetic Algorithm* ou simplesmente pela sigla SGA. Nele, trabalha-se com uma população fixa, cujas cadeias de caracteres estão binariamente codificadas (GOLDBERG, 1989). Após estudar o problema a ser otimizado, deve-se definir qual a quantidade de indivíduos que terá a população, a formação cromossômica do indivíduo e as pro-

babilidades de aplicação dos operadores genéticos. Sua aplicação na resolução de um problema deve seguir os passos a seguir:

1. Iniciar uma população, de tamanho *SIZE_POP*, com cromossomos gerados aleatoriamente;

2. Aplicar a função de adequação em cada cromossomo da população;

3. Criar novos cromossomos através dos cruzamentos dos cromossomos selecionados desta população. Aplicar recombinação e mutação nestes cromossomos;

4. Eliminar membros da antiga população a fim de inserir novos cromossomos, mantendo a população com o mesmo número *SIZE_POP* de cromossomos;

5. Aplicar a função de adequação aos novos cromossomos e inseri-los na população;

6. Se a solução ideal for encontrada, ou se o tempo (ou número de gerações) se esgotar, retornar o cromossomo com a melhor adequação. Caso contrário, voltar ao passo (3).

Se tudo ocorrer bem, esta simulação do processo evolutivo irá produzir, à medida que as gerações forem se sucedendo, cromossomos cada vez mais bem adaptados, isto é, com melhor valor da função de adequação, de maneira que, no final, obtém-se uma solução (cromossomo) com alto grau de adequação ao problema proposto.

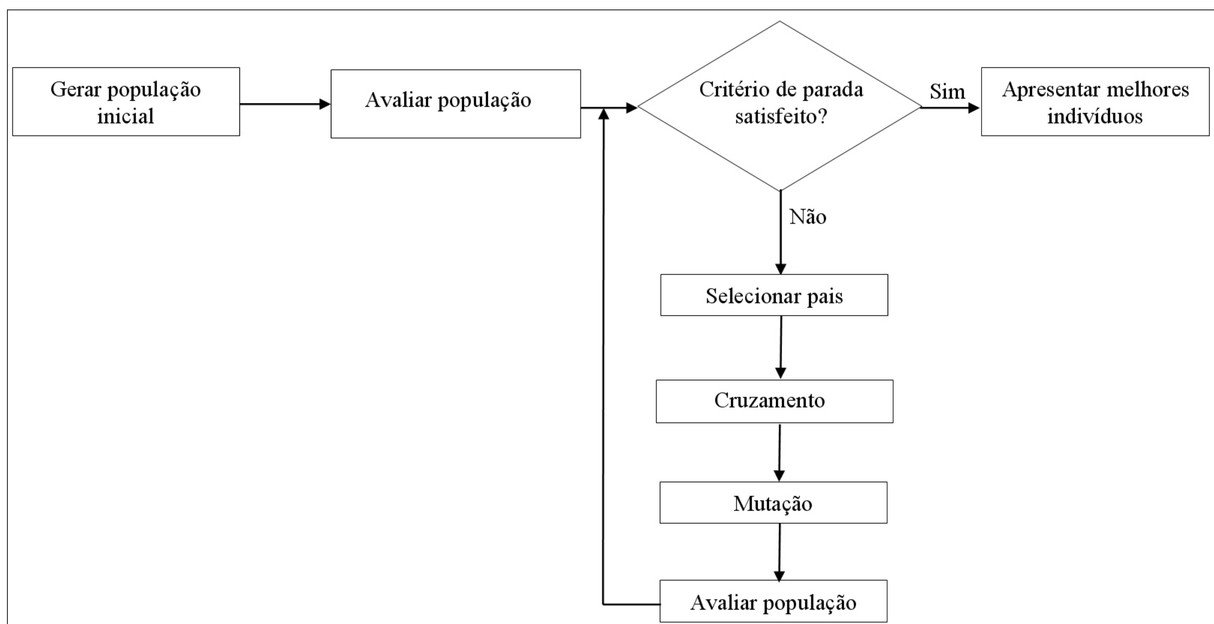


Figura 2.1: Fluxograma de um Algoritmo Genético Simples.

A Figura 2.1 mostra o fluxograma de um Algoritmo Genético Simples, contendo os princípios básicos de evolução da população de indivíduos através do tempo, a aplicação do

critério de seleção dos indivíduos mais bem adaptados e o uso dos operadores de cruzamento e de mutação.

O Algoritmo Genético, a partir de agora referenciado neste trabalho simplesmente por AG, tem além das suas características originais, um número grande de variações propostas na literatura. A Seção 2.4 apresenta, para as fases que compõem os AGs, algumas destas variações.

2.4 Representação do Cromossomo

Como mencionado anteriormente, os AGs são baseados na Biologia dos seres vivos. Biologicamente falando, um cromossomo é uma seqüência de DNA que contém vários genes (vide Figura 2.2). O gene é um dos fatores que determinam a forma ou função de uma ou várias características dos seres vivos, como por exemplo: a cor dos olhos, altura, cor da pele, etc. O gene ocupa um lugar na estrutura do cromossomo ((locus)) e pode ter uma dentre várias formas alternativas (alelo). Alelos diferentes, de um mesmo gene, em cromossomos distintos ocupam o mesmo (locus). O conjunto de genes de um indivíduo forma o seu genótipo e, conseqüentemente, diversas de suas características. No entanto, o meio onde o indivíduo se desenvolve influencia diretamente na formação do mesmo, definindo assim o seu fenótipo. O fenótipo é que define a constituição do indivíduo, sua aparência física, a manifestação específica de uma característica, etc.

Figura 2.2: Cromossomo e seqüência de DNA com gene em destaque.

A representação do cromossomo é fundamental para o AG e consiste de uma maneira de modelar a informação do problema em um formato viável de ser tratado pelo computador. Em AGs, um cromossomo é uma estrutura de dados que representa uma das possíveis soluções do espaço de busca de um problema. Quanto mais a representação for adequada ao problema melhores serão os resultados obtidos (LINDEN, 2006) (GOLDBARG; LUNA, 2000).

A definição inadequada da codificação pode levar a problemas de convergência prematura do Algoritmo Genético(VIANNA; OCHI; DRUMMOND, 1999). A estrutura de um cromossomo deve representar uma solução como um todo e deve ser a mais simples possível. Em problemas de otimização restrita, a codificação adotada pode fazer com que indivíduos modificados por cruzamento/mutação sejam inválidos. Nestes casos, cuidados especiais devem ser tomados na definição da codificação e/ou dos operadores.

Os AGs, na sua formulação original, possuem o alfabeto binário como forma de representar uma solução em um espaço de busca (GOLDBERG, 1989). Contudo, tanto o método de representação quanto o alfabeto genético podem variar de problema para problema. (ANTONISSE, 1989) e (RADCLIFFE, 1992) afirmam que boas propriedades dos AGs não são bem aproveitadas se a técnica binária de representação for utilizada para determinados problemas. Isto motiva a implementação de AGs utilizando representação não binária, para resolver de forma "mais natural" problemas específicos. Alguns exemplos de estruturas utilizadas na literatura são: vetores de números reais, vetores de números inteiros, matrizes bidimensionais, listas encadeadas, etc (HERRERA; LOZANO; VERDEGAY, 1998). Em um AG, o termo indivíduo é aplicado a cada membro da população. Cada indivíduo possui um ou mais cromossomos (genótipo), formado(s) por uma estrutura de dados, que contém uma representação de uma possível solução para o problema (fenótipo). Os cromossomos são compostos por genes, isto é, caracteres que compõem o conteúdo da estrutura de dados. O gene possui um valor (alelo) que se encontra em uma posição ((locus)) da estrutura de dados. A Figura 2.3 apresenta uma tabela comparativa entre os termos da genética natural e dos AGs.

Genética Natural	Algoritmo Genético
gene	caractere (parte formadora da estrutura)
alelo	valor do caractere
cromossomo	estrutura de dados
locus	posição do caractere na estrutura
genótipo	estrutura(s) de dados que formam o indivíduo
fenótipo	estrutura(s) de dados decodificadas

Figura 2.3: Tabela comparativa: Genética Natural X Algoritmo Genético.

Dependendo de como um problema está codificado para ser trabalhado por um AG, para a obtenção da sua solução (fenótipo), deverá ser feita uma decodificação da representação do indivíduo. Este fato será mais amplamente abordado na Seção 2.5.

2.5 Avaliação do Indivíduo

Em uma população natural, quanto mais adaptado, maiores serão as chances do indivíduo sobreviver no ambiente e reproduzir-se, passando parte de seu material genético às gerações posteriores. Nos AGs, a função de aptidão (*fitness*) associa a cada indivíduo da população uma medida de sua qualidade, representando quão bem adaptado o indivíduo está. Os indivíduos com melhores valores para a função de aptidão terão, futuramente, mais chances de serem selecionados para participar dos processos de cruzamentos e transmitir as suas características.

O conceito de função de aptidão está intimamente ligado a idéia de função objetivo. Contudo, a função objetivo fornece uma medida de qualidade em relação a um problema específico e é restrita a um indivíduo, enquanto a função de aptidão transforma essa medida em uma grandeza que representa oportunidade de reprodução. Além disso, a aptidão é sempre definida em relação aos demais indivíduos da população.

Algumas vezes o valor da aptidão pode ser fornecido pela própria função objetivo. Isso, porém, nem sempre é possível. Se os valores fornecidos por essa função forem muito próximos uns dos outros, por exemplo, o processo evolutivo corre o risco de tornar-se aleatório, dependendo do esquema de seleção adotado. Um outro problema que pode ocorrer é a aptidão de um indivíduo ser muito maior do que as demais e dominar a população no momento da seleção, gerando uma convergência prematura.

Percebe-se ainda que os AGs trabalham tentando propiciar o desenvolvimento de indivíduos, cujo valor de aptidão esteja acima da média, até chegar àqueles de mais alto desempenho. Portanto, nota-se que os AGs trabalham em termos de maximização. No entanto, muitas vezes, o objetivo é minimizar uma função. Em casos como esse, também é necessário transformar a função objetivo em uma função de aptidão, para que esse problema seja corrigido. Outros cuidados com a função objetivo incluem evitar que ela retorne valores negativos e respeitar as restrições impostas pelo problema de otimização (SOARES, 1997).

2.6 População Inicial

Para que o AG inicie um processo evolutivo, deve ser gerada de alguma forma uma população inicial. O número de indivíduos dessa população, que em muitos algoritmos se mantém fixo durante todo o processo evolutivo, é considerado um parâmetro desses algoritmos e sua determina-ção é feita de modo empírico. A geração dos indivíduos da população inicial pode ser feita de diferentes maneiras, que dependem fortemente da forma de representação

adotada.

Um método comumente utilizado na criação da população é a inicialização aleatória dos indivíduos (VIANNA; OCHI; DRUMMOND, 1999). Se algum conhecimento inicial a respeito do problema estiver disponível, este pode ser utilizado na inicialização da população. Por exemplo, se é sabido que a solução final (assumindo codificação binária) vai apresentar mais 0's do que 1's, então esta informação pode ser utilizada, mesmo que não se saiba exatamente a proporção. Já em problemas com restrições, deve-se tomar cuidado para não gerar indivíduos inválidos na etapa de inicialização.

Entretanto, em muitas aplicações a utilização do método de criação aleatória gera uma população com indivíduos de pouca qualidade, o que exige dos operadores genéticos muito esforço (muitas gerações) para atingir uma boa solução. Nesses casos é mais eficiente usar heurísticas construtivas aleatorizadas para a criação da população inicial.

2.7 Seleção de Indivíduos

A etapa de seleção tem por objetivo escolher os indivíduos que deverão sofrer a ação de operadores genéticos e, através destes, compor uma nova geração. Apesar de existirem métodos aleatórios, boa parte dos métodos de seleção são criados para selecionar, preferencialmente, indivíduos com melhores valores para a função de aptidão, embora indivíduos menos favorecidos algumas vezes sejam selecionados a fim de manter a diversidade da população.

O Algoritmo Genético Simples proposto por Holland (HOLLAND, 1975) utiliza a Roleta como método de seleção de indivíduos para cruzamento. Este método determina uma probabilidade maior de escolha dos melhores indivíduos para cruzamento, permitindo assim a evolução das futuras gerações. O método de seleção da roleta será descrito com maiores detalhes na Subseção 2.7.1 da mesma forma que outras políticas encontradas em (GOLDBERG, 1989) (BLICKLE; THIELE, 1995) (BRINDLE, 1981) (BAKER, 1987).

2.7.1 Roleta (Seleção Estocástica com Reposição - Stochastic Sampling with Replacement)

Este método foi proposto por (HOLLAND, 1975) para o seu primeiro AG e seu funcionamento se assemelha à roleta dos cassinos. Uma vez calculada a função de aptidão para os cromossomos da população, cada indivíduo terá a oportunidade de ser selecionado para cruzamento de acordo com o seu desempenho relativo ao do grupo. Supondo f_i o valor da função de aptidão do i-ésimo cromossomo, a probabilidade do indivíduo i ser selecionado será dada pela

razão $f_i/\sum f$, onde $\sum f$ é o somatório do valor das funções de aptidão (adequação) de todos os indivíduos da população.

No exemplo da Figura 2.4, dados quatro cromossomos com valores 30, 44, 36 e 21 para função de adequação, um gráfico representando a roleta pode ser observado.

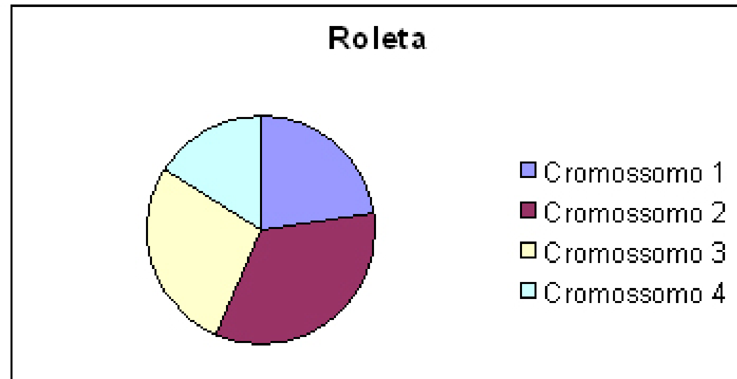


Figura 2.4: Exemplo de uma Roleta para Seleção de Indivíduos.

De acordo com a Figura 2.4, pode-se observar que o setor obtido por um cromossomo representa uma maior ou menor probabilidade do mesmo ser selecionado após o giro da roleta. Neste exemplo, o cromossomo 2 é o que possui a maior probabilidade de ser selecionado, enquanto o cromossomo 4 é o de menor probabilidade.

A roleta será girada tantas vezes quantas forem necessárias para obter o número requerido de indivíduos para o cruzamento. O número de indivíduos selecionados a cada geração será de acordo com os diferentes tipos de AGs. Como os indivíduos cujas regiões possuem maior área terão maior probabilidade de serem selecionados, a seleção de indivíduos pode conter várias cópias de um mesmo indivíduo enquanto outros podem desaparecer. Logo, este método apresenta tendência de convergência prematura podendo direcionar a solução para um ótimo local.

2.7.2 Seleção Aleatória (Stochastic Universal Sampling - SUS)

Neste método, a escolha é feita aleatoriamente entre membros da população. Cada indivíduo tem a chance de 1/*SIZE_POP* de ser escolhido, sendo *SIZE_POP* o número total de indivíduos da população. Assim, todos os indivíduos possuem a mesma probabilidade de serem selecionados. Esta forma de seleção possui uma probabilidade muito remota de causar a evolução da população sobre a qual atua (BAKER, 1987).

2.7.3 Torneio (Variação Aleatória)

Neste método um subconjunto de indivíduos (normalmente um par) é escolhido aleatoriamente. Contudo, somente o indivíduo com maior valor para a função de adequação é selecionado para participar do cruzamento (BLICKLE; THIELE, 1995). Este processo é repetido até que a quantidade de indivíduos selecionados seja a necessária para a realização do cruzamento.

Quanto maior for o subconjunto de indivíduos escolhidos, maior será a velocidade com que os indivíduos mais fortes dominarão a população e, conseqüentemente, extinguirão os mais fracos.

2.7.4 Torneio (Variação Roleta)

É uma política similar ao Torneio, porém o método de obtenção do subconjunto de indivíduos para a realização do torneio é o da Roleta (no lugar do método aleatório)(GOLDBERG, 1989) (BRINDLE, 1981).

2.7.5 Seleção Determinística (Deterministic Sampling - DS)

O funcionamento desta política de seleção pode ser dividido em duas partes. Primeiramente, é gerada uma população temporária e, em seguida, os indivíduos desta população são selecionados de forma randômica para participar do cruzamento.

A população temporária é gerada da seguinte forma: (1) Divide-se o valor da função de aptidão de um indivíduo pela média das funções de aptidão dos demais indivíduos da população original (f_i/f_{med}). (2) A parte inteira do resultado determina o número de cópias do indivíduo i que fará parte da população temporária. Devido à existência de partes fracionárias provenientes dos cálculos, a população não será totalmente preenchida (vide Figura 2.5).

(3)Para completar a população, deve-se ordenar os indivíduos da população original de forma decrescente, utilizando como índice a parte fracionária da função de adequação de cada indivíduo. A partir deste ponto os indivíduos de maior parte fracionária são utilizados para preencher as vagas restantes da população (GOLDBERG, 1989) (BRINDLE, 1981).

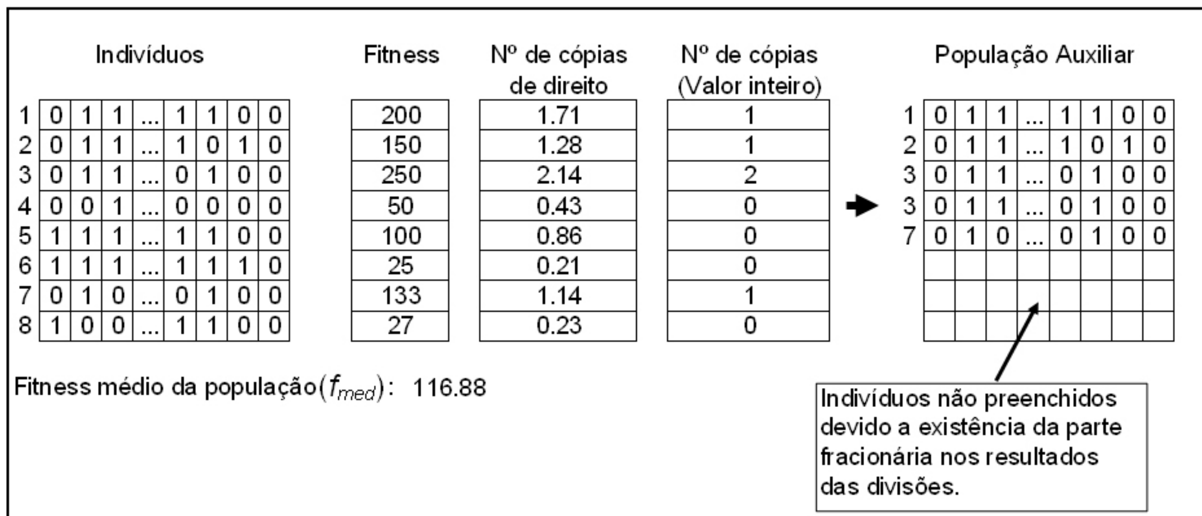


Figura 2.5: Exemplo de Geração de uma População Auxiliar.

2.7.6 Seleção Estocástica por Resto Sem Reposição (Stochastic Remainder Sampling)

Método semelhante à Seleção Determinística, porém as vagas restantes na população temporária (vide Figura 2.5) são preenchidas de forma diferente. A parte fracionária do cálculo da expectativa de cópias de um indivíduo é utilizada como valor de probabilidade na seleção deste. Por exemplo, um indivíduo que tenha como resultado do cálculo f_i/f_{med} o valor 2.14 terá 2 cópias garantidas na população temporária e 14% de chance de, caso seja escolhido, possuir uma terceira cópia. O indivíduo a ter sua possibilidade de seleção avaliada deverá ser escolhido por um critério (o aleatório, por exemplo) e em seguida um número aleatório entre 1 e 100 deverá ser gerado. Caso este número seja menor ou igual a 14 (14% de probabilidade), uma cópia do indivíduo é adicionada à população, caso contrário um outro indivíduo é escolhido aleatoriamente e o processo se repete até que seja completada a população(GOLDBERG, 1989) (BRINDLE, 1981).

2.7.7 Seleção Estocástica por Resto Com Reposição (Stochastic Remainder Sampling with Replacement)

Método semelhante aos dois anteriores, porém as vagas restantes na população temporária (vide Figura 2.5) são preenchidas diferentemente. As partes fracionárias dos cálculos das expectativas de cópias dos indivíduos, são utilizadas como valores para os setores de uma roleta (GOLDBERG, 1989) (BRINDLE, 1981). A seleção por roleta é executada até que as vagas da população temporária sejam completadas.

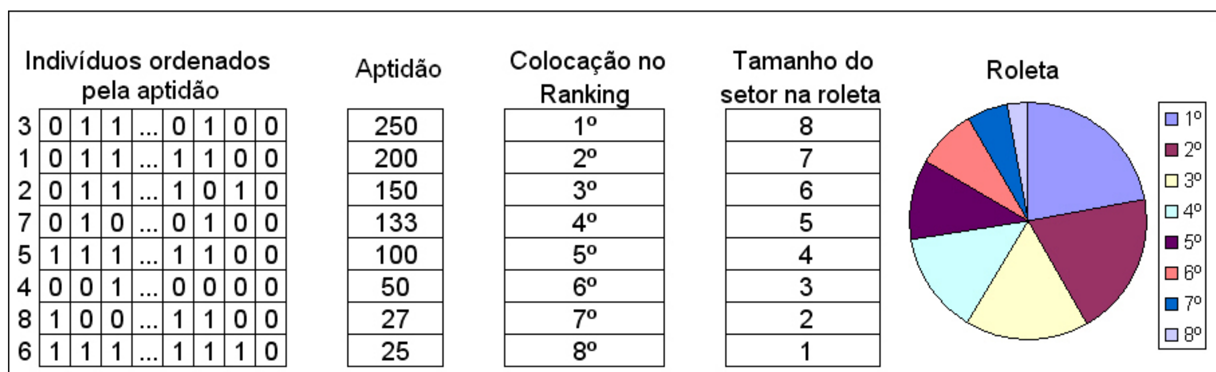
2.7.8 Seleção Estocástica Sem Reposição (Stochastic Sampling without Replacement)

Neste método é calculado o número de vezes que um indivíduo em potencial fará parte de cruzamentos para gerar descendentes em uma nova população. Este cálculo é feito dividindo-se o valor da função de aptidão de um indivíduo pela média das funções de aptidão dos demais indivíduos da população original (f_i/f_{med}). Caso o resultado seja um número inteiro, o mesmo é armazenado em um vetor que guarda a quantidade de cópias dos indivíduos. Caso o resultado seja fracionário, primeiro aproxima-se o valor para o inteiro subseqüente para depois fazer o armazenamento no vetor.

Uma vez tendo o vetor de cópias preenchido, o método da roleta é utilizado e, à medida que os indivíduos forem selecionados para reprodução, os números de cruzamentos permitidos para estes são decrementados até que cheguem a zero e os indivíduos sejam retirados da roleta (JONG, 1975) (GOLDBERG, 1989) (BRINDLE, 1981).

2.7.9 Seleção por *ranking*

Inicialmente, os indivíduos da população são ordenados de acordo com os valores das suas funções de aptidão. Após a ordenação, cada indivíduo terá um valor equivalente à sua posição no *ranking* e, um procedimento similar à seleção pelo método da roleta é utilizado (vide Figura 2.6). Quanto melhor a posição do indivíduo no *ranking*, maior será seu setor na roleta e conseqüentemente maior a sua chance de ser selecionado (BAKER, 1987).



Indivíduos ordenados pela aptidão		Aptidão	Colocação no Ranking	Tamanho do setor na roleta
3	0 1 1 ... 0 1 0 0	250	1º	8
1	0 1 1 ... 1 1 0 0	200	2º	7
2	0 1 1 ... 1 0 1 0	150	3º	6
7	0 1 0 ... 0 1 0 0	133	4º	5
5	1 1 1 ... 1 1 0 0	100	5º	4
4	0 0 1 ... 0 0 0 0	50	6º	3
8	1 0 0 ... 1 1 0 0	27	7º	2
6	1 1 1 ... 1 1 1 0	25	8º	1

Figura 2.6: Exemplo de Seleção por *Ranking*.

2.8 Cruzamento

O cruzamento (*crossover*) é o operador responsável pela recombinação de características dos indivíduos durante a reprodução, permitindo que os filhos herdem características dos pais. É considerado o operador genético mais importante (LINDEN, 2006) e ocorre com uma probabilidade definida por uma taxa de cruzamento (*pc*), que comumente varia de 60% a 95% . Uma taxa baixa pode significar pouco aproveitamento da informação existente; já uma taxa alta pode provocar convergência prematura (homogeneização rápida da população), pois o cruzamento associado à reprodução ajuda a uniformizar a população.

O Algoritmo Genético Simples proposto por Holland (HOLLAND, 1975) utiliza o cruzamento com um ponto de corte para gerar novos indivíduos. A seguir esta política será apresentada juntamente com outras comumente encontradas na literatura.

2.8.1 Cruzamento com um Ponto de Corte

Para que um cruzamento seja realizado, um ponto da estrutura do cromossmo é escolhido aleatoriamente. Baseado neste ponto, o cromossomo é dividido em duas partes (não necessaria-mente do mesmo tamanho). O primeiro filho é formado juntando-se a primeira parte do primeiro indivíduo com a segunda parte do segundo indivíduo; já o segundo será formado com as partes restantes, como pode ser visto na Figura 2.7.

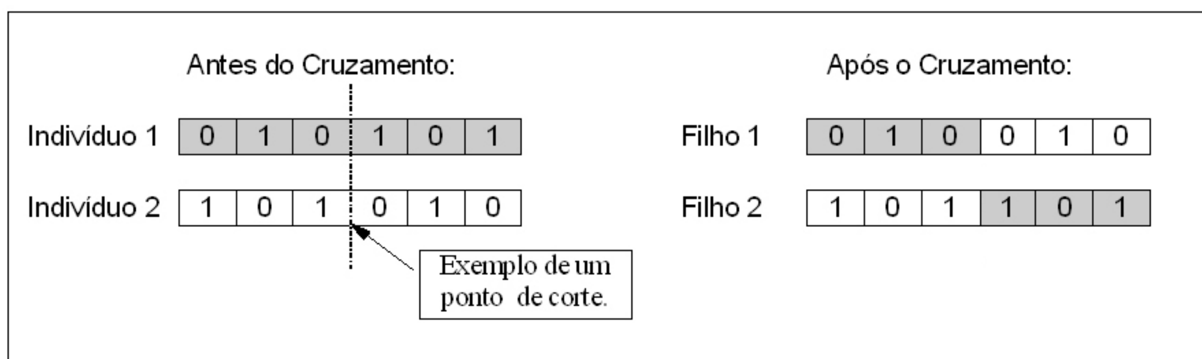


Figura 2.7: Exemplo de Cruzamento com Um Ponto de Corte.

2.8.2 Cruzamento com *N* Pontos de Corte

Similar ao cruzamento com um ponto, os pontos de corte são escolhidos aleatoriamente. Se algum ponto for sorteado mais de uma vez, não se procura por outro. Os locais de corte do indivíduo podem variar de 1 a L-1, sendo L o comprimento total do indivíduo. Um exemplo de cruzamento com N pontos de corte pode ser visto na Figura 2.8.

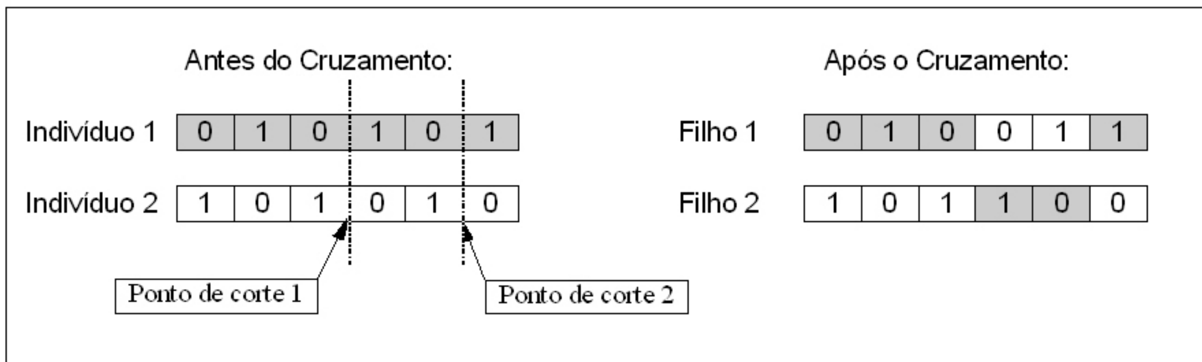


Figura 2.8: Exemplo de Cruzamento com *N* Pontos de Corte (onde *N*=2).

2.8.3 Cruzamento Uniforme

Nesse operador, para cada gene dos indivíduos, verifica-se a ocorrência de um evento com probabilidade de 50%. Caso afirmativo, o gene corrente do primeiro pai é copiado no primeiro filho e o gene corrente do segundo pai é copiado no segundo filho. Caso negativo, o inverso ocorre, isto é, o gene corrente do primeiro pai é copiado no segundo filho e o gene corrente do segundo pai é copiado no primeiro filho.

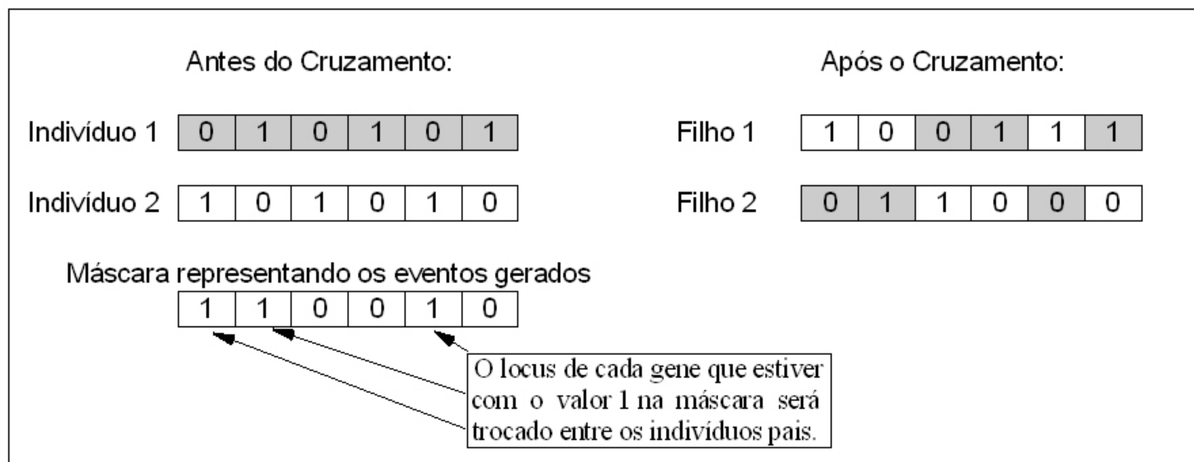


Figura 2.9: Exemplo de Cruzamento Uniforme.

A Figura 2.9 exemplifica este método utilizando cromossomos binários. Neste exemplo foram selecionados dois indivíduos para cruzamento, com seis genes cada. A fim de representar os eventos gerados para cada gene foi criada uma máscara com seis posições. Cada posição está relacionada a um *(*locus) dos indivíduos selecionados. Para cada posição da máscara é gerado aleatoriamente um número 0 ou um número 1. Caso seja gerado o número 1, os genes correspondentes àquele *(*locus) (posição) nos indivíduos participantes do cruzamento, são trocados. Caso o número gerado seja o 0, os genes correspondentes àquele *(*locus) permanecem nos seus indivíduos de origem. Na Figura 2.9 são apresentados os filhos gerados através do cruzamento

uniforme, utilizando o exemplo representado na máscara. Nesta figura foram trocados os genes existentes nos *(*locus) 1, 2 e 5.

2.8.4 Cruzamento Baseado em Ordem

Os métodos de cruzamento baseados em ordem, também conhecidos como recombinação, foram projetados para problemas de ordenação, como arranjos e permutações. Nesses casos, não deve ser permitido que o cruzamento gere cromossomo com elementos repetidos. Essa restrição inviabiliza os procedimentos de cruzamentos descritos anteriormente para problemas como caixeiro viajante, coloração em grafo, dentre outros desta categoria. Os operadores de cruzamento baseados em ordem OX, CX e PMX aparecem com freqüência na literatura e podem ser melhor estudados em (GOLDBERG, 1989) , (MICHALEWICZ, 1994) e (OLIVER; SMITH; HOLLAND, 1987).

2.9 Mutação

Após o cruzamento entra em ação o operador de mutação. Este tem com principais funções a inserção de novas características e a restauração de material genético perdido nos processos de seleção e cruzamento, objetivando assim a introdução e manutenção da diversidade genética da população. Desta forma, a mutação assegura uma probabilidade de examinar qualquer ponto do espaço de busca visando contornar problemas de ótimos locais.

O operador de mutação é aplicado aos indivíduos com uma probabilidade dada pela taxa de mutação *pm*. Se *pm* for muita baixa pode acontecer um comprometimento da diversidade na população. Por outro lado, se *pm* for muito alta, acontecerão muitas perturbações aleatórias e os filhos provavelmente começarão a perder suas semelhanças com os pais, podendo comprometer a convergência. Linden (2006) (LINDEN, 2006) afirma que o valor da probabilidade que determina se o operador de mutação será ou não aplicado é um dos parâmetros dos AGs que apenas a experiência pode determinar; porém é um valor comumente baixo.

Alguns exemplos de valores fixados pela literatura são: *pm*=0.001 (JONG, 1975), *pm*=0.01 (GREFENSTETTE, 1986), *pm*=1/*L* (onde *L* é o tamanho do cromossomo) e *pm*=1.75/(*L***SIZE_POP*) (onde *SIZE_POP* é o tamanho da população) (BÄCK, 1991). A implementação da mutação, quando se usa o código binário, consiste apenas na mudança do valor do bit escolhido (vide Figura 2.10).

O operador de mutação percorre todos os genes do cromossomo e, para cada gene, gera um evento com probabilidade *pm*; se este evento ocorrer, o valor do gene é trocado.

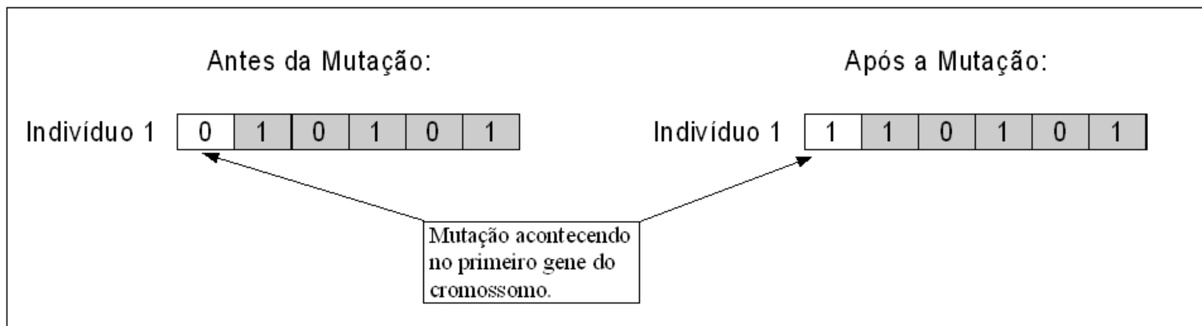


Figura 2.10: Exemplo de Mutação.

2.10 Evolução da População

Nesta etapa é definida a porcentagem de indivíduos substituídos em cada iteração após o método de cruzamento (GOLDBERG; DEB, 1989) (LOZANO; HERRERA; CANO, 2008) (VASCONCELOS; TAKAHASHI; SALDANHA, 2001).

2.10.1 Simple Genetic Algorithm (SGA)

Proposto por (HOLLAND, 1975) foi a primeira forma de implementação de AGs. Nesta, a população de uma geração é completamente substituída pela geração seguinte. A evolução da população se dá através dos operadores de seleção, cruzamento e mutação. Na etapa de seleção os indivíduos mais bem adaptados possuem maior chance de serem escolhidos para cruzamento. Este fato contribui para a evolução da população, porém não impede que algum indivíduo mais bem adaptado de uma geração possa ser substituído por outro de uma geração futura, gerando um retrocesso na população.

Uma variante do algoritmo SGA é o SGA com elitismo. O elitismo é uma estratégia de implementação onde os *N* melhores indivíduos de cada geração são preservados para a geração seguinte (LINDEN, 2006). Isto garante que o melhor indivíduo de uma nova geração seja pelo menos igual ao melhor indivíduo da geração anterior. Apesar de normalmente colaborar para a melhoria do desempenho do AG, esta técnica pode gerar convergência prematura para uma solução ótima local.

2.10.2 Steady State Genetic Algorithm (SSGA)

Este algoritmo difere-se do SGA principalmente em relação à substituição da população (CEDENO; VEMURI; SLEZAK, 1995) (SYSWERDA, 1989). No SSGA, usualmente, somente um ou dois descendentes são produzidos em cada geração. Após o cruzamento, são

definidos os indivíduos da população que serão substituídos pelos novos descendentes (LOZANO; HERRERA; CANO, 2008). Dentre as políticas de substituição comumente encontradas na literatura estão a substituição dos piores indivíduos, substituição aleatória, substituição dos indivíduos mais semelhantes e substituição por algum critério de classificação (SOARES, 1997) (CEDENO; VEMURI; SLEZAK, 1995).

2.10.3 Parâmetro Gap da Geração

O Gap da geração é um parâmetro que controla a porcentagem da população a ser substituída em cada geração (CARBONO; MENEZES, 2005) (GREFENSTETTE, 1986). Nos SGAs este parâmetro é igual a 100%, o que significa que todos os indivíduos da população serão substituídos. Nos SSGA, este valor é usualmente 1/*SIZE_POP*, quando apenas um indivíduo é substituído (onde *SIZE_POP* é o tamanho da população utilizada). Da mesma forma que o algoritmo SSGA, quando utilizado o parâmetro Gap, têm que ser definidos os indivíduos da população que serão substituídos pelos novos descendentes.

2.11 Critério de Parada

Na etapa de avaliação do critério de parada é verificado se o AG deve continuar ou não evoluindo sua população. O Algoritmo Genético simples proposto por Holland (HOLLAND, 1975) utiliza como critério de parada um número máximo de gerações. Além deste, outras políticas são descritas em (AIEX; RESENDE; RIBEIRO, 2003) (GOLDBERG, 1989) (CHOU; PREMKUMAR; CHU, 1999) (RIBEIRO; VIANNA, 2003) (SOARES, 1997) (VASCONCELOS et al., 1997), e são explicadas nas próximas subseções.

2.11.1 Número Máximo de Gerações

O número máximo de gerações é um parâmetro definido pelo usuário. Quando o AG alcança este valor, o algoritmo é finalizado. O número ideal de gerações deve ser definido após a execução de testes para detectar a convergência do algoritmo (GOLDBERG, 1989) (VASCONCELOS et al., 1997).

2.11.2 Melhora Mínima da Solução em um Determinado Número de Gerações

O melhor valor de função objetivo encontrado em uma geração é subtraído do melhor valor encontrado até a geração anterior. Se este cálculo não exceder um valor determinado pelo

usuário, durante um certo número de gerações, é porque não ocorreu melhora e o algoritmo é finalizado.

2.11.3 Convergência da População

O melhor valor encontrado para a função objetivo é subtraído da média dos valores das funções objetivo de todos os indivíduos da população. Se este cálculo não exceder um valor predeterminado pelo usuário, a população convergiu, e o algoritmo é finalizado.

2.11.4 Tempo de Computação

Nesta política, o usuário define o tempo máximo de computação para a execução do algoritmo. Após decorrido este tempo, o algoritmo é finalizado e a melhor solução retornada (CHOU; PREMKUMAR; CHU, 1999).

2.11.5 Valor Alvo

Um valor alvo para a função objetivo é definido pelo usuário. O algoritmo finaliza quando, em uma geração, o cálculo da função objetivo de um dos indivíduos da população resultar em um valor maior ou igual ao alvo pré-definido (AIEX; RESENDE; RIBEIRO, 2003) (RIBEIRO; VIANNA, 2003), no caso de problemas de maximização,ou um valor menor ou igual ao alvo, no caso de problemas de minimização.

2.12 Parâmetros dos Algoritmos Genéticos

Um dos problemas que deve ser abordado por quem pretende usar um AG é a escolha dos seus parâmetros. A maioria dos AGs utiliza três parâmetros: Tamanho da População (*SIZE_POP*), Probabilidade de Cruzamento (*pc*), e Probabilidade de Mutação(*pm*).

A teoria sobre AG não auxilia muito sobre a escolha de seus parâmetros, uma vez que tais configurações dependem do problema abordado. Usualmente são utilizados valores testados empiricamente (SOARES, 1997). Na literatura são encontrados diversos estudos sobre o ajuste dos parâmetros do AG entre os quais pode-se destacar : (JONG, 1975), (GREFENSTETTE, 1986), (GOLDBERG, 1989) e (SCHAFFER et al., 1989).

A eficiência e funcionamento dos AGs são altamente dependentes dos seus parâmetros de controle, cujos tipos básicos são descritos a seguir.

2.12.1 Tamanho da População (*SIZE_POP*)

Uma população pequena possui amostragem insuficiente do espaço de busca do conjunto solução para a maioria dos problemas, podendo conduzir o algoritmo na direção de um ótimo local. Já uma população grande contém quantidade bem representativa do conjunto solução, e também, acarreta uma convergência mais lenta da população, possibilitando aos AGs explorarem mais a informação existente. Entretanto, para uma população grande, o número de cálculos de função desempenho por geração pode resultar em um tempo computacional inaceitável. Na literatura são encontrados valores como: 50 a 100 indivíduos (JONG, 1975), 20 a 30 indivíduos (SCHAFFER et al., 1989) e 30 indivíduos (GREFENSTETTE, 1986).

2.12.2 Probabilidade de Cruzamento (*pc*)

Esse parâmetro controla a freqüência com a qual o operador de cruzamento é aplicado. Valores baixos de *pc* significam pouco aproveitamento da informação existente e lentidão para a convergência do algoritmo. Já os valores altos de *pc* introduzem novos indivíduos mais rapidamente e podem provocar convergência prematura (homogeneização rápida da população), pois o cruzamento ajuda a uniformizar a população. Na literatura são encontrados valores como: 0,6 (JONG, 1975), 0,75 a 0,95 (SCHAFFER et al., 1989) e 0,95 (GREFENSTETTE, 1986).

2.12.3 Probabilidade de Mutação (*pm*)

A probabilidade de mutação indica a taxa em que haverá a mutação de cromossomos nas populações ao longo da evolução. A mutação é realizada gene a gene segundo uma probabilidade *pm*. Pequenos valores de *pm* não proporcionam o devido aumento da diversidade populacional, limitando assim o espaço de busca. Por outro lado, um alto valor de *pm* conduz os AGs à procura aleatória. Na literatura são encontrados valores como: 0,001 (JONG, 1975), 0,005 a 0,001 (SCHAFFER et al., 1989) e 0,01 (GREFENSTETTE, 1986).

3 Trabalhos Relacionados

Diversos trabalhos encontrados na literatura visam a construção de ferramentas para facilitar a utilização de algoritmos genéticos. Alguns tem por objetivo complementar aplicativos já existentes, como o pacote GEATBX (POHLHEIM, 2005) para o software Matlab (da área de computação científica) e a ferramenta Evolver (CORPORATION, 2001) para o processador de planilhas Microsoft Excel; outros são bibliotecas ou pacotes de software que, adicionados aos códigos fonte dos programas do usuário, reduzem o trabalho de implementação e modificação dos algoritmos genéticos, como a biblioteca GALib (WALL, 1996), o pacote Genesis (GREFENSTETTE, 1990), o sistema GALOPPS (GOODMAN, 1994) e a ferramenta TOGAI, proposta neste trabalho. Estas bibliotecas e pacotes de software, por terem maior relação com este trabalho, serão apresentados a seguir.

A biblioteca GALib é gratuita, de código fonte aberto e multiplataforma, isto é, pode ser executada em sistemas operacionais distintos, tais como Unix, Mac/OS e DOS/Windows. É uma das mais completas bibliotecas de software para implementação de AGs, pois fornece um grande conjunto de objetos para serem utilizados com a linguagem de programação C++. A GALib possibilita utilizar tipos distintos de seleção, *crossover* e mutação em diferentes representações cromossomiais, além de usar diversos módulos de população e critérios de parada distintos. Novos algoritmos genéticos podem ser rapidamente testados desde que derivados da base genética dos algoritmos presentes na biblioteca. Por ser desenvolvida em C++ utilizando técnicas de orientação a objetos, esta biblioteca limita o seu universo de usuários a profissionais com grande experiência nesta área de programação.

O Genesis é um AG baseado em gerações escrito na linguagem de programação C por John Grefenstette (GREFENSTETTE, 1990). Como foi um dos primeiros pacotes distribuídos de forma gratuita e extensa, o Genesis foi muito influente na popularização dos algoritmos genéticos e vários pacotes disponíveis atualmente são baseados nele. No Genesis, a representação cromossômica do indivíduo pode ser feita utilizando apenas as estruturas de dados vetor de números binários e vetor de números reais. O número de políticas diferentes para implementação das etapas dos AGs é reduzido comparado com as demais ferramentas aqui apresentadas.

GALOPPS (*Genetic ALgorithm Optimized for Portability and Paralelism*) é um sistema escrito na linguagem C que cria um AG baseado nos princípios do AG mais simples. Para usá-lo, o usuário deve definir uma função de avaliação e quaisquer outras funções necessárias, usando modelos (*templates*) definidos pelo sistema. Pode rodar em uma ou mais subestações, possuindo um modo texto (*batch*) ou um modo gráfico (somente para sistema operacional SOLARIS) para maior interação com o usuário. Apesar de bem diversificado no número de políticas e utilizar representação binária e não binária, o sistema é limitado a cromossomos formados por vetores. Já foi testado nos ambientes DOS, Windows, NeXTStep, SunOS, Solaris, HP-UX e Macintosh.

Com objetivos semelhantes às demais ferramentas, a TOGAI, proposta neste trabalho e descrita com maiores detalhes no Capítulo 4, se diferencia por focar nos mecanismos que compõem os AGs e são independentes do problema, isto é, podem ser utilizados na implementação não importando o domínio da aplicação. A TOGAI possui interface gráfica implementada na Linguagem Java e gera o Algoritmo Genético em código C ANSI, ganhando desta forma portabilidade para a ferramenta e desempenho de linguagem compilada para o AG gerado. Além disso, a linguagem C ANSI foi escolhida por não demandar conhecimentos de programação orientada a objetos, facilitando desta forma o uso da ferramenta por pesquisadores não especialistas na área de informática e aumentando assim o número de potenciais usuários.

A TOGAI, na sua primeira versão, disponibiliza diversas políticas de seleção, critério de parada e evolução da população, permitindo que AGs gerados pela ferramenta sejam facilmente modificados com o intuito de buscar as melhores políticas a serem utilizadas com o problema. Da mesma forma que as demais ferramentas aqui analisadas, a TOGAI permite que novas políticas geradas pelo usuário sejam inseridas na ferramenta possibilitando, assim, o seu crescimento. No Capítulo 4 outras características e formas de uso da TOGAI são descritas com mais profundidade, visando ao seu entendimento e utilização.

4 A Ferramenta TOGAI (Tool for Genetic Algorithms Implementation)

Os Algoritmos Genéticos possuem como característica ter seus códigos fonte e parâmetros intimamente ligados aos problemas que visam solucionar. Desta forma, percebe-se que uma vez gerada uma implementação para um problema, para que a mesma seja reaproveitada para um problema futuro, um grande trabalho de reimplementação pode ser necessário. Antonisse (ANTONISSE, 1989) e Radcliffe (RADCLIFFE, 1992) sugerem o uso de uma representação mais real da estrutura cromossômica para alguns problemas objetivando um melhor aproveitamento das propriedades dos AGs. Com isso, estruturas de dados distintas precisam ser utilizadas em diferentes problemas. Este fato reitera a dificuldade de reaproveitamento de código para implementações de problemas distintos, podendo tornar inviável este reaproveitamento.

A dificuldade na determinação de parâmetros e políticas também é característica nos AGs. Dada uma implementação de AG, resultados apresentados em testes computacionais para um problema podem se tornar bastante diferentes, caso sejam alterados parâmetros ou políticas de implementação. Como já referenciado, políticas e parâmetros dos AGs muitas vezes são definidos para um problema após a realização de testes que apresentem a viabilidade e o ganho gerado pelos seus usos.

Visando a geração de facilitadores para as dificuldades expostas, foi criada a ferramenta TOGAI (*Tool for Genetic Algorithms Implementation*). Esta se propõe, via interface gráfica, a gerar uma implementação parcial de Algoritmo Genético, contendo toda a codificação considerada independente das especificações do problema abordado. Do mesmo modo que viabiliza o uso de diversas políticas de implementação de AGs para serem testadas e avaliadas. Será função do usuário complementar o código com a programação específica e dependente do problema. A ferramenta se caracteriza ainda por permitir que políticas implementadas pelo usuário sejam adicionadas à mesma, gerando assim um aumento do número de possibilidades de seu uso.

Para que a TOGAI seja executada é necessário que estejam disponíveis no computador uma máquina virtual *JAVA* e um compilador *C ANSI*. Como estes estão disponíveis para diversas plataformas, então pode-se considerar a ferramenta multiplataforma. Este fato contribui para a

difusão e para o crescimento da mesma, pois sendo multiplataforma, o universo de usuários em potencial é maior.

A TOGAI foi desenvolvida utilizando a linguagem de programação *JAVA*, devido a sua portabilidade e robustez. Uma das características da ferramenta é possuir uma interface gráfica, que tem por função receber os parâmetros e escolhas (políticas) do usuário, que servem de subsídios para a geração do código do AG. Este código foi desenvolvido na linguagem de programação *C ANSI* devido às suas características de portabilidade e desempenho de linguagem compilada. O modelo de codificação de AG gerado pela ferramenta divide a implementação em dois grandes blocos: o código dependente das características do problema e o código independente das características do problema.

4.1 O modelo de Desenvolvimento dos Algoritmos Genéticos

A transformação de problemas em formatos viáveis de serem utilizados por AGs, pode gerar representações cromossômicas distintas. Estas, para serem codificadas necessitam de tipos e estruturas de dados também distintas. As diversas estruturas de dados que podem fazer parte da representação do cromossomo de um AG, necessitam de diferentes especificações de código para manipulá-las. Desta forma, parte do código de um AG é completamente dependente da estrutura de dados utilizada, inviabilizando um possível uso desta codificação para um problema representado por uma estrutura distinta.

O modelo de implementação utilizado tem como função principal separar a codificação do AG dependente da estrutura de dados daquela que é independente, isto é, separar a parte do código de um AG que pode ser reaproveitada para diversos domínios de problema, da parte que é dependente do problema e que para problemas com representações distintas exigem modificações. Desta forma, cria-se uma estrutura modulada onde as etapas dos AGs que são independentes do problema não mais necessitarão de implementações, ficando como função para o desenvolvedor somente as etapas dependentes do problema.

Devido às características acima citadas, é importante deixar a implementação da estrutura do cromossomo para o usuário. Assim, os algoritmos desenvolvidos não terão sua eficiência prejudicada por alguma restrição do modelo.

A Figura 4.1 apresenta a estrutura de um indivíduo da população dentro do modelo proposto, codificada na linguagem *C ANSI*. A estrutura *tpChromossome* define a estrutura interna do cromossomo. A codificação desta estrutura é de responsabilidade do usuário, que deve estruturá-la da maneira que achar mais conveniente para resolução do problema em questão. A estrutura *tpIndivid* é definida pelo modelo. Esta possui um dado do tipo *tpChromossome*, repre-

sentando o cromossomo que define o indivíduo, e um dado que representa o valor de avaliação deste indivíduo. Este dado é do tipo *tpObj* que é, por padrão, associado ao *float* (tipo básico da linguagem *C ANSI*, que armazena valores pertencentes ao conjunto dos números reais), mas pode ser modificado pelo usuário do sistema, quando necessário.

```
1  /* Estrutura do cromossomo */
2  typedef struct chromossome
3  {
4      /* Deverá ser incluída aqui a estrutura de dados do cromossomo
5      Ex:  O cromossomo poderá ser formado por 10 elementos inteiros:
6      int iAlele[10];*/
7  } tpChromossome;
8
9  /* Deverá ser definido aqui o tipo de dado que guarda o valor da função objeti
10 /* Tipo padrão: float. */
11 typedef float tpObj;
12
13 /* Estrutura dos indivíduos: formada pelos cromossomos e pelos valores de suas*/
14 typedef struct individual
15 {
16     tpChromossome chrom;
17     tpObj obj;
18 } tpIndivid;
19
20 /* Definição do tipo da população como um vetor de individuos */
21 typedef tpIndivid tpPopulation[SIZE_POP];
```

Figura 4.1: Definição, na linguagem *C ANSI*, das estruturas que formam um indivíduo.

Algumas etapas do AG são dependentes da estrutura cromossômica (*tpChromossome*), ou seja, precisam conhecer a estrutura interna do cromossomo para realizar uma consulta ou modificação. As principais são a geração da população inicial e as operações de cruzamento e mutação.

O modelo proposto tem como objetivo diminuir o trabalho do usuário ao desenvolver um Algoritmo Genético para um determinado problema. Além disso, o modelo torna a implementação do AG organizada (modulada) o suficiente para evitar o “retrabalho” com a modificação de certos métodos. Por exemplo, uma vez implementado um AG, é possível testar diferentes políticas de seleção de indivíduos (já implementadas e disponíveis pelo modelo) neste AG sem a necessidade de novas codificações. Isso é possível, pois os métodos de seleção de indivíduos, de evolução da população e os critérios de parada são freqüentemente dependentes da estrutura *tpIndivid* (Figura 4.1), sem a necessidade de acessar a estrutura interna do cromossomo.

Nas próximas subseções serão apresentadas as etapas dos AGs, bem como suas implementações dentro do modelo.

4.1.1 População Inicial

Como mencionado anteriormente, a população inicial é formada por indivíduos que representam possíveis soluções para o problema, mas que, via processo evolutivo, sofrerão a ação dos operadores genéticos objetivando as suas transformações em soluções melhores.

Dentro do modelo de implementação proposto, esta etapa é formada por dois procedimentos: o que cria a população inicial e o que cria um indivíduo.

O procedimento que cria a população inicial (Figura 4.2) é independente do problema, isto é, não precisa lidar com a estrutura do cromossomo em sua implementação. Este procedimento utiliza o tamanho da população como critério de parada da sua estrutura de repetição (Figura 4.2, Linha 4), onde são feitas as chamadas ao procedimento de criação de indivíduo. O tamanho da população é identificado pela constante *SIZE_POP* e deve ter o seu valor alterado pelo usuário de acordo com a sua necessidade.

```
1 void initialPopulation( tpPopulation pop )
2 {
3     int idIndiv;   /* Identificador do indivíduo da população*/
4     for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
5     {
6         createIndivid( &pop[idIndiv] );
7     }
8 }
```

Figura 4.2: Procedimento que gera a população inicial.

O procedimento que cria indivíduo é dependente do problema, isto é, depende da estrutura interna do cromossomo (Figura 4.1). Nesta o indivíduo da população é construído, podendo o mesmo ser composto de qualquer tipo de estrutura de dados, quer seja estática ou dinâmica. O usuário é responsável pela codificação desta função, mas deve respeitar o cabeçalho definido pelo modelo para a criação do indivíduo. Este cabeçalho está definido a seguir:

*void createIndivid(tpIndivid *pIndivid)*

em que **pIndivid* é um ponteiro com o endereço onde será alocado o novo indivíduo da população.

4.1.2 Avaliação do Indivíduo

Após gerado um indivíduo é necessário calcular a aptidão do mesmo, isto é, o seu *fitness*. Através deste, o indivíduo terá uma medida de qualidade, representando quão bem adaptado está, definindo assim, um parâmetro de comparação deste com os demais.

Para que a aptidão do indivíduo seja calculada, é necessário que a estrutura de dados que forma o cromossomo seja conhecida e utilizada. Desta forma o procedimento que calcula a aptidão é dependente do problema, isto é, depende da estrutura interna do cromossomo (Figura 4.1). O usuário é responsável pela codificação desta etapa, mas deve respeitar o cabeçalho definido pelo modelo para a criação do indivíduo. Este cabeçalho está definido a seguir:

*void calcFitness(tpIndivid *pChildren)*

em que **pChildren* é um ponteiro que aponta para a estrutura do indivíduo que terá o seu *fitness* calculado.

Devido ao fato do cálculo da aptidão do indivíduo ser feito após a etapa de cruzamento (exceto para a população inicial), o ponteiro passado por parâmetro no cabeçalho da função *calcFitness* foi identificado por *pChildren*, o que caracteriza o indivíduo como filho dentro daquela geração.

4.1.3 Seleção de Indivíduos

Uma vez criada uma nova geração, caberá aplicar os operadores genéticos a uma parte dos indivíduos da população visando a criação de novos indivíduos com melhores características genéticas e conseqüentemente melhores valores das suas funções de aptidão. Para que o procedimento anterior tenha sucesso, os indivíduos participantes do processo precisam ser escolhidos cuidadosamente, permitindo assim que a evolução aconteça e ao mesmo tempo impedindo a homogeneidade da população.

Todas as políticas de seleção apresentadas no Capítulo 2 (Seção 2.7) dependem unicamente dos valores das funções de aptidão dos indivíduos da população para serem implementadas, o que está definido na estrutura *tpIndivid* (Figura 4.1). Desta forma, não são necessários conhecimentos a respeito da estrutura de dados que forma o cromossomo e conseqüentemente o indivíduo. Este fato possibilita que sejam desenvolvidas e incluídas na ferramenta políticas de seleção que podem ser utilizadas nos mais diversos problemas sem que modificações no código sejam necessárias.

Para isso, todas as políticas devem obedecer ao cabeçalho definido pelo modelo para os métodos de seleção. Este cabeçalho está definido a seguir:

*void selection(tpPopulation pop, int *pSelected)*

onde *pop* é a estrutura que armazena os cromossomos e as aptidões dos indivíduos da população e **pSelected* é o vetor que é retornado com a identificação (índice) dos indivíduos selecionados.

As políticas de seleção citadas neste trabalho foram implementadas e estão disponíveis para serem utilizadas com a ferramenta TOGAI, permitindo assim que o usuário possa escolher a que se adapta melhor ao problema a ser resolvido.

Um exemplo de codificação de uma política de seleção, de acordo com o modelo, é apresentado na Figura 4.3. O método exemplificado é o método do torneio (variação aleatória) com número de competidores (*TOT_COMPETIT*) igual a três, descrito na seção 2.7.3. A fim de facilitar o entendimento, o pseudocódigo desta política é explicado e apresentado na Figura 4.4.

```
1  #define TOT_COMPETIT 3 /* Total de competidores para participar do torneio */
2  void selection( tpPopulation pop, int* pSelected )
3  {
4      int    idIndiv,       /* Identificador do indivíduo da população*/
5             idNewFather,   /* Indice que referencia os indivíduos selecionados (no
6             iCompetitor,   /* Indice que referencia os competidores sorteados*/
7             iBetterIndiv;  /* Melhor indivíduo do subconjunto sorteado */
8      tpObj betterObj;      /* Armazena o melhor valor de função objetivo */
9
10     /* Seleção dos indivíduos de acordo com o número de pais a serem
11        selecionados */
12     for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
13     {
14         /* Sorteio de um competidor e definição inicial do mesmo como o melhor *
15         idIndiv = rand(  ) % SIZE_POP;
16         betterObj = pop[idIndiv].obj;
17         iBetterIndiv = idIndiv;
18         /* Sorteio dos demais competidores e execução das comparações (torneio)
19         for ( iCompetitor = 1; iCompetitor < TOT_COMPETIT; iCompetitor++ )
20         {
21             idIndiv = rand(  ) % SIZE_POP;
22             if ( best( pop[idIndiv].obj > betterObj ) )
23             {
24                 betterObj = pop[idIndiv].obj;
25                 iBetterIndiv = idIndiv;
26             }
27         }
28         /* Atribuição de um novo indivíduo selecionado à estrutura que será
29            retornada */
30         pSelected[idNewFather] = iBetterIndiv;
31     }
32 }
```

Figura 4.3: Procedimento que seleciona indivíduo para cruzamento utilizando a política torneio (variação aleatório).

O *Procedimento seleção* (vide Figura 4.4) recebe por parâmetro as estruturas de dados *População* e *VetSelecionado*. O parâmetro *População* armazena todos os indivíduos da população e é utilizado pelo procedimento na busca dos indivíduos mais adaptados. O parâmetro *VetSelecionado* é utilizado para armazenar os indivíduos que forem selecionados para cruzamento (vide Linha 1). Uma estrutura de repetição executa um número de iterações igual ao número de filhos que farão parte do cruzamento (vide Linha 2). Para cada filho a ser selecionado, um número de indivíduos, igual ao valor da constante *total_de_competidores*, é escolhido aleatoriamente para participar do torneio. É incluído em *VetSelecionado* o participante mais

apto do torneio.

```
1  Procedimento Seleção( População, VetSelecionado ) { Torneio variação Aleatório }
2      Para o número de indivíduos a serem selecionados Faça
3          MelhorPai <- Indivíduo aleatório da população;
4          MelhorFitness <- Fitness do indivíduo selecionado aleatoriamente;
5          Para total_de_competidores - 1 Faça
6              PaiCandidato <- Indivíduo aleatório da população;
7              Se Fitness_PaiCandidato > MelhorFitness Então
8                  MelhorFitness <- Fitness_PaiCandidato
9                  MelhorPai <- PaiCandidato;
10             Fim-Se
11         Fim-Para
12         Armazenar MelhorPai no vetor VetSelecionado {vetor de pais selecionados};
13     Fim-Para
14 Fim-Seleção
```

Figura 4.4: Algoritmo que seleciona indivíduo para cruzamento utilizando a política torneio (variação aleatório).

4.1.4 Cruzamento

O operador de cruzamento ou recombinação cria novos indivíduos através da combinação de dois ou mais indivíduos. A idéia intuitiva por trás do operador de cruzamento é a troca de informação entre diferentes soluções candidatas. Para que isto seja possível é necessário conhecer a estrutura de dados que forma o cromossomo, pois o cruzamento se dá, dependendo do problema, através da troca de informações existentes nestas estruturas ou até mesmo pela própria alteração destas.

Devido ao fato de ser uma etapa dependente da estrutura interna do cromossomo (Figura 4.1), o usuário fica responsável pela codificação desta, porem este deve respeitar o cabeçalho definido pelo modelo para a etapa de cruzamento. Este cabeçalho está definido a seguir:

*void crossover(tpPopulation pop, int *pSelected, tpIndivid *pChildren)*

em que *pop* é a estrutura que armazena os cromossomos e as funções objetivo dos indivíduos da população, **pSelected* é o vetor que possui a identificação dos indivíduos que foram selecionados para o cruzamento e **pChildren* é o vetor que armazena os filhos gerados no cruzamento.

4.1.5 Mutação

O operador de mutação modifica aleatoriamente um ou mais genes de um cromossomo de acordo com uma certa probabilidade. A idéia por trás do operador de mutação é criar uma variabilidade extra na população, mas sem destruir o progresso já obtido com a busca.

A alteração de um gene, dependendo do problema, implica na alteração de uma informação

existente na estrutura de dados ou até mesmo na alteração desta. Deste modo, também pode-se perceber que a etapa de mutação é dependente da estrutura interna do cromossomo (Figura 4.1) e deve ser codificada pelo usuário respeitando-se o cabeçalho definido pelo modelo a seguir:

*void mutation(tpChromossome *pChromChild)*

em que **pChromChild* é um ponteiro que aponta para a estrutura do cromossomo de um descendente, gerado no último cruzamento, que sofrerá a mutação.

4.1.6 Evolução da População

Após a aplicação dos operadores genéticos de cruzamento e mutação é necessária a inclusão dos novos indivíduos gerados, na nova população. Nesta etapa é definida a porcentagem de indivíduos que serão substituídos em cada geração.

As políticas de evolução da população, apresentadas no Capítulo 2 (Seção 2.10), foram implementadas para serem utilizadas pelo usuário na ferramenta e não necessitam de inclusão ou alteração de código por parte deste. Porém, para que um indivíduo seja copiado de uma estrutura de dados para outra, é feita uma chamada a função que copia indivíduo, cuja implementação deverá ser feita pelo usuário, da forma apresentada na Seção 4.2.2.

Todas as políticas de evolução da população aqui implementadas, possuem uma função principal cujo cabeçalho obedece ao seguinte modelo proposto:

tpPopulation populationEvolution(tpPopulation pop, tpIndivid *bestIndivid)*

em que *pop* é a estrutura que armazena os cromossomos e funções objetivo dos indivíduos da população e *bestIndivid* é a estrutura que armazena a melhor solução até o momento.

Um exemplo de codificação de uma política de evolução da população, de acordo com o modelo, é apresentado na Figura 4.5. O método exemplificado é o SGA (descrito na Subseção 2.10.1). Neste pode-se perceber que a política de evolução da população controla quantos indivíduos sofrerão a ação dos operadores genéticos de cruzamento e mutação a cada geração e a inclusão dos seus descendentes na nova população. Desta forma, as chamadas aos procedimentos que implementam estes operadores são feitas de dentro do procedimento *populationEvolution* (Figura 4.5, Linhas 19 e 32), bem como os controles de suas probabilidades (Figura 4.5, Linhas 17 e 30) e a seleção dos indivíduos (Figura 4.5, Linha 14).

```
 1 tpPopulation* populationEvolution( tpPopulation pop, tpIndivid *bestIndivid )
 2 {
 3       int         idChild,               /* Índice de um filho gerado */
 4                   idNewIndivid = 0,      /* Índice do novo indivíduo criado */
 5                   iProb;                 /* Número randômico gerado para checagem
 6       int         selected[TOT_SELECT];  /* vector com a identificação dos indivíd
 7       tpIndivid   children[TOT_CHILDREN];/* vector com os indivíduos gerados em u
 8       tpIndivid   *newPop;               /* Ponteiro para a nova população (válid
 9
10       newPop = ( tpIndivid* ) malloc ( SIZE_POP * sizeof( tpIndivid ) );
11       while ( idNewIndivid < SIZE_POP )
12       {
13           /* Aloca a estrutura que armazenará os indivíduos selecionados */
14           selection ( pop, selected );
15           /* Cruzamento e sua probabilidade */
16           iProb=rand(  ) % 100;
17           if( ( PROB_CROSS * 100 ) >= iProb )  /*Verifica se o cruzamento deve s
18           {
19                crossover( pop, selected, children );
20           }
21           else
22           {
23               /* Esta função fará com que os pais sejam mantidos para a próxima
24               keepFather( pop, selected, children );
25           }
26           /* Probabilidade de mutação */
27           for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
28           {
29              iProb=rand(  ) % 100;
30              if( ( PROB_MUTAT * 100 ) >= iProb ) /* Verifica se a mutação deve s
31              {
32                   mutation( &children[idChild].chrom );
33              }
34           }
35           for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
36           {
37                   verifyChromChild( &children[idChild].chrom ); /* Checar a viab
38                   calcObj( &children[idChild] );                /* Calcula o val
39                   bestSolut( &children[idChild], bestIndivid ); /* Atualiza a me
40           }
41           updatePop( newPop, children, &idNewIndivd ); /* Atualiza a população */
42
43       }
44    deallocPopulation( pop );  /* Desaloca a estrutura auxiliar utilizada */
45    return newPop;
46 }
```

Figura 4.5: Procedimento que implementa a evolução da população SGA.

Para melhor entendimento, o pseudocódigo para a política de evolução da população SGA é apresentado na Figura 4.6.

O *Procedimento EvoluçãoPopulação* (vide Figura 4.6) recebe por parâmetro as estruturas de dados *População* e *MelhorIndivíduo*. O parâmetro *População* armazena todos os indivíduos da população, enquanto o parâmetro *MelhorIndivíduo* armazena o indivíduo cuja função de adaptação é a melhor dentre todas as gerações. A estrutura de repetição existente na Linha 2, define que o código nela contido será executado até que toda a população seja renovada. Para cada iteração são realizados: (1) A seleção dos pais que participam do cruzamento (vide Linha 3), (2) um cruzamento com uma certa probabilidade (vide Linhas 4 a 8), (3) a mutação

dos indivíduos gerados, com uma certa probabilidade (vide Linhas 10 a 12), (4) a validação dos indivíduos gerados (vide Linha 13), (5) o cálculo das funções de aptidão dos indivíduos (vide Linha 15), (6) a verificação se a aptidão do indivíduo é a melhor dentre todas as calculadas (vide Linha 15) e (7) a atualização da população com os filhos gerados (vide Linha 17). Após a criação da nova geração, a população é retornada para o procedimento que controla o critério de parada do algortimo (vide Linha 19).

```
1  Função EvoluçãoPopulação( População, MelhorIndivíduo )
2      Para o número de indivíduos a serem criados Faça
3          Seleção ( População, VetSelecionado ); { Selecionar pais }
4          Se evento ocorrer com probabilidade de cruzamento pc Então
5              Realizar cruzamento entre os pais selecionados;
6          Senão
7              Manter os pais selecionados para a próxima geração;
8          Fim-Se
9          Para cada filho gerado Faça
10             Se evento ocorrer com probabilidade de mutação pm Então
11                 Realizar mutação no filho gerado;
12             Fim-Se
13             Validar o filho gerado quanto a viabilidade da solução;
14             Calcular o fitness do filho gerado;
15             Verificar se o filho gerado é a melhor solução até o momento;
16         Fim-Para
17         Atualizar a população com os filhos gerados;
18     Fim-Para
19     Retornar a nova população;
20 Fim-EvoluçãoPopulação
```

Figura 4.6: Algoritmo que implementa a evolução da população SGA.

As políticas de evolução da população implementadas para a ferramenta, de acordo com o modelo, executam a atualização da população através de uma chamada ao procedimento *updatePop* (Figura 4.7). Este tem por objetivo substituir os antigos indivíduos por aqueles criados na nova geração. É neste procedimento que é feita a chamada ao procedimento que copia indivíduos de uma estrutura de dados para outra, como comentado anteriormente.

```
1  void updatePop( tpPopulation newPop, tpIndivid *pChildren, int *idNewIndivid )
2  {
3      int idChild; /* Índice de um filho gerado */
4      for( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
5      {
6          copyIndivid ( &(newPop[*idNewIndivid]), &(pChildren[idChild]));
7          (*idNewIndivid)++;
8          if ( *idNewIndivid == SIZE_POP )
9          {/*Se toda a população já tiver sido trocada encerra a atualização*/
10             break;
11         }
12     }
13 }
```

Figura 4.7: Procedimento que atualiza a população.

Outros procedimentos chamados pelo procedimento de evolução da população são apre-

sentados na Seção 4.2.2.

4.1.7 Critério de Parada

Uma vez atualizada a população, chega o momento de verificar se é necessária a criação de uma nova geração de indivíduos. O procedimento que implementa o critério de parada determina, no modelo, se novas gerações serão criadas ou se é o momento de terminar a execução do algoritmo.

Os critérios de parada apresentados no Capítulo 2 (Seção 2.11) dependem dos valores das funções de aptidão dos indivíduos da população ou dependem de tempo computacional para serem implementadas. Desta forma, não é necessário conhecimento a respeito da estrutura de dados que forma o cromossomo e, conseqüentemente, o indivíduo. Este fato possibilita que sejam desenvolvidos e incluídos na ferramenta critérios de parada que podem ser utilizados nos mais diversos problemas, sem que modificações no código sejam necessárias.

Para isso, todas as políticas devem obedecer ao cabeçalho definido pelo modelo para os critérios de parada. Este cabeçalho está definido a seguir:

tpPopulation stopCriterion (tpPopulation pop, tpIndivid *bestIndivid)*

em que *pop* é a estrutura que armazena os cromossomos e as aptidões dos indivíduos da população e **bestIndivid* é o ponteiro que aponta para a estrutura do melhor indivíduo gerado pelo Algoritmo Genético.

Os critérios de parada citados neste trabalho foram implementados e estão disponíveis para serem utilizados com a ferramenta TOGAI, permitindo assim que o usuário possa escolher o que se adapta melhor ao problema a ser resolvido.

Um exemplo de codificação de critério de parada, de acordo com o modelo, é apresentado na Figura 4.8. O critério exemplificado é o número máximo de gerações.

```
1  tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid )
2  {
3     int  iGenerat;/* Identificador da geração */
4
5     iGenerat = 0;
6     while ( iGenerat < MAX_GEN ) /*Controla o número total de gerações*/
7     {
8        pop = populationEvolution ( pop, bestIndivid );
9        iGenerat++;
10    }
11    return pop;
12 }
```

Figura 4.8: Procedimento que implementa o critério de parada número máximo de gerações.

4.2 Detalhes de Implementação do Modelo

Além da implementação das principais etapas de um AG, apresentadas na seção anterior, foram necessárias codificações de funções complementares, constantes e estratégias para formar o modelo. Esta seção visa apresentar estas codificações mostrando seus objetivos e forma de uso.

4.2.1 Constantes

Constantes que armazenam parâmetros comumente definidos em AGs foram declaradas e podem ter os seus valores modificados, visando adequar a implementação ao problema do usuário. No modelo foram utilizadas as seguintes constantes:

- *SIZE_POP*: Constante que armazena o número de indivíduos existentes na população. Foi definido o valor 100 como padrão para esta constante.
- *TOT_SELECT*: Constante que armazena o número de indivíduos que serão selecionados para fazer parte da etapa de cruzamento. Foi definido o valor 2 como padrão para esta constante.
- *TOT_CHILDREN*: Constante que armazena o número de filhos gerados após a etapa de cruzamento. Foi definido o valor 2 como padrão para esta constante.
- *PROB_MUTAT*: Constante que armazena o valor que será utilizado como probabilidade na etapa de mutação. Foi definido o valor 0.01 (1%) como padrão para esta constante.
- *PROB_CROSS*: Constante que armazena o valor que será utilizado como probabilidade na etapa de cruzamento. Foi definido o valor 0.8 (80%) como padrão para esta constante.

4.2.2 Funções Complementares

Além das funções que implementam etapas específicas dos AGs foram necessárias as implementa-ções dos procedimentos apresentados a seguir.

Função Desalocar População (*deallocPopulation*)

A função *deallocPopulation*, apresentada na Figura 4.9, tem por objetivo possibilitar ao usuário desalocar estruturas de dados que porventura tenham sido utilizadas para formar os indivíduos da população.

```
1  void deallocPopulation( tpPopulation pop )
2  {
3      int idIndiv;  /* Identificador do indivíduo da população*/
4
5      for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
6      {
7          deallocIndivid( &pop[idIndiv] );
8      }
9      free(pop);
10 }
```

Figura 4.9: Função que desaloca a população.

Para cada indivíduo da população é chamada a função *deallocIndivid*, que visa desalocar um indivíduo que tenha sido criado com uma estrutura de dados dinâmica. Como esta função lida diretamente com a estrutura de dados do indivíduo, deverá ser codificada pelo usuário respeitando o cabeçalho definido no modelo como se segue:

*void deallocIndivid(tpIndivid *pIndivid)*

onde **pIndivid* é um ponteiro com o endereço do indivíduo que será desalocado da memória.

Função Copiar Indivíduo (*copyIndivid*)

A função *copyIndivid*, apresentada na Figura 4.10, tem por objetivo copiar os campos que formam um indivíduo de uma estrutura de dados para outra.

```
1 void copyIndivid( tpIndivid *pDestin, tpIndivid *pOrigin )
2 {
3     copyChrom(&(pDestin->chrom), &(pOrigin->chrom));
4     pDestin->obj = pOrigin->obj;
5 }
```

Figura 4.10: Função que copia indivíduo.

No modelo inicial, a função que atualiza população (*updatePop*), utiliza a função *copyIndivid* para substituir indivíduos por seus descendentes em uma população (políticas SSGA e GAP) ou para copiar indivíduos gerados em cruzamentos para a nova população (política SGA). No entanto, esta função pode ser de grande utilidade para o usuário na implementação da codificação específica do problema abordado.

Como a estrutura do indivíduo é composta pelos campos cromossomo e função objetivo, e como a estrutura de dados do cromossomo é dependente do problema abordado, é

necessário que o usuário codifique a cópia do cromossomo que é identificada no modelo pela função *copyChrom*. Esta deve ser implementada respeitando-se o cabeçalho definido no modelo como segue:

*void copyChrom(tpChromossome *pDestin, tpChromossome *pOrigin)*

em que **pDestin* é o ponteiro com o endereço da estrutura do cromossomo que será substituído e **pOrigin* é o ponteiro com o endereço da estrutura do cromossomo que será copiado.

Função Manter Pais (*keepFather*)

Na etapa de cruzamento, indivíduos selecionados têm os seus genes combinados visando a geração de novos indivíduos descendentes. Contudo, este processo é realizado em função da probabilidade de cruzamento. Caso o cruzamento não ocorra, a nova população deverá ser preenchida, possivelmente, com os indivíduos que haviam sido selecionados para cruzamento.

A função *keepFather* existente no modelo e apresentada na Figura 4.11, tem por objetivo manter os melhores indivíduos, dentre os selecionados para o cruzamento, na geração seguinte. Quando, na etapa de cruzamento, o número de descendentes a serem gerados é menor do que o número de pais, a função *keepFather* insere os pais um a um na nova população, em ordem decrescente em relação ao valor de sua função de aptidão(*fitness*), até que todos os espaços que estavam disponibilizados na nova população, para aquele cruzamento, sejam preenchidos. Porém, se o número de descendentes for maior do que o número de pais, a função *keepFather* também insere os pais um a um na nova população, em ordem decrescente em relação ao valor de sua aptidão, mas uma vez que todos os pais sejam inseridos, o processo se reinicia até que todos os espaços disponibilizados sejam alocados. Caso o número de pais seja igual ao número de descendentes, cada pai será copiado para a nova população, preenchendo os espaços que estavam destinados aos seus descendentes.

A função *keepFather* depende somente dos valores das funções de aptidão dos indivíduos da população para seu funcionamento. Desta forma, não é necessário conhecimento a respeito da estrutura de dados que forma o cromossomo e conseqüentemente o indivíduo. Apesar de estar implementada na íntegra e não necessitar de codificação por parte do usuário, esta pode ser modificada desde que o cabeçalho proposto pelo modelo seja mantido como segue:

*void keepFather(tpPopulation pop, int *pSelected, tpIndivid *pChildren)*

em que *pop* é a estrutura que armazena os cromossomos e funções objetivo dos indivíduos da população, **pSelected* é o vetor que será retornado com a identificação dos indivíduos selecionados e **pChildren* é o ponteiro que aponta para a estrutura onde estão os

```
1  void keepFather( tpPopulation pop, int *pSelected, tpIndivid *pChildren )
2  {
3      int    idChild,  /* Identificação do filho na estrutura de filhos gerada pelo
4             idFather, /* Identificação do pai na estrutura de pais gerada pela sel
5             idBetter; /* Identificação do melhor indivíduo em um conjunto */
6      tpObj betterObj,/* Melhor valor de função objetivo dentre um grupo de indiví
7             prevObj;  /* Melhor valor de função objetivo computada anteriormente p
8
9      prevObj = OBJ_VAL_END;
10     /* Loop controlado pelo número de filhos que serão, na verdade, preenchidos
11     for ( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
12     {
13         betterObj = OBJ_VAL_INI;
14         /* Loop controlado pelo número de indivíduos anteriormente selecionados
15         for ( idFather = 0; idFather < TOT_SELECT; idFather++ )
16         {
17             /* Comparações para encontrar o melhor pai dos ainda não selecionado
18             if  (  best( pop[pSelected[idFather]].obj, betterObj ) )
19             {
20                 if ( best( prevObj, pop[pSelected[idFather]].obj ) )
21                 {
22                     betterObj = pop[pSelected[idFather]].obj;
23                     idBetter = pSelected[idFather];
24                 }
25             }
26         }
27         prevObj = betterObj;
28
29         pChildren[idChild].obj = betterObj;
30         /* Loop para copiar o cromossomo da estrutura do pai para o filho */
31
32         copyChrom( &pChildren[idChild].chrom, &pop[idBetter].chrom );
33
34         /* Se o número de filhos for maior do que o de pais o processo reinicia
35         if ( idChild == idFather - 1 )
36         {
37             prevObj = OBJ_VAL_END;
38         }
39     }
40 }
```

Figura 4.11: Função para manter pais na nova geração.

filhos gerados por um cruzamento.

Visando um melhor entendimento, um pseudocódigo para a função *keepFather* será descrito e apresentado na Figura 4.12.

O *Procedimento ManterPais* (vide Figura 4.12) recebe por parâmetro as estruturas de dados *População*, *VetSelecionado* e *VetFilhos* (vide Linha 1). O parâmetro *População* armazena todos os indivíduos da população e será utilizado para que, de posse do índice de um indivíduo selecionado, tenha-se acesso a sua estrutura de dados. O parâmetro *VetSelecionado* armazena a identificação dos pais, anteriormente selecionados, que participariam do cruzamento. Devido a não ocorrência do cruzamento, um conjunto destes pais é copiado para a estrutura de dados *VetFilhos*, permanecendo assim na geração seguinte.

A estrutura de repetição codificada na Linha 3 é definida para que o número de iterações

```
1  Procedimento ManterPais ( População, VetSelecionado, VetFilhos )
2      número_de_pais_copiados <- 0
3      Para o número de filhos que deveriam ser gerados no cruzamento Faça
4          Encontrar o melhor pai, p, dos selecionados para cruzamento e
           ainda não copiados para VetFilhos
5          Copiar p para o vetor VetFilhos;
6          número_de_pais_copiados <- número_de_pais_copiados + 1
7          Se  número_de_pais_copiados = número_de_pais_do_cruzamento Então
8              Se número_de_filhos > número_de_pais_do_cruzamento Então
9                  número_de_pais_copiados <- 0
10                 Reiniciar copiando o melhor pai para o vetor VetFilhos
11             Fim-Se
12         Fim-Se
13     Fim-Para
14 Fim ManterPais
```

Figura 4.12: Algoritmo para manter pais na nova geração.

seja igual ao número de filhos que deveriam ser gerados, permitindo assim que, devido a não ocorrência do cruzamento, os lugares que seriam ocupados pelos filhos sejam preenchidos com os próprios indivíduos selecionados. Para cada iteração encontra-se o pai mais apto, dentre aqueles que não foram selecionados para permanecer na futura geração (vide Linha 4). Em seguida, o pai encontrado é copiado para o vetor *VetFilhos* (vide Linha 5). Foram utilizadas duas estruturas de seleção, a primeira verifica se todos os pais já foram contemplados com uma cópia para a geração seguinte (vide Linha 7), a segunda verifica se o número de filhos que deveriam ser gerados é maior do que a quantidade de indivíduos selecionados para cruzamento (pais). Caso todos os pais já tenham sido contemplados com uma cópia para a geração seguinte, e o número de pais seja igual ao número de filhos que deveriam ser gerados, o algoritmo é encerrado. Se o número de filhos que deveriam ser gerados for maior do que o número de pais, então a variável que controla o número de pais copiados deverá ser inicializada para que os pais novamente sejam copiados para as lacunas deixadas pelos filhos até que todos os espaços reservados para os descendentes, resultantes do cruzamento que não aconteceu, sejam preenchidos.

4.3 Utilização da Ferramenta TOGAI

Utilizando dos conceitos e do modelo descrito na Seção 4.1 foi criada a ferramenta TOGAI (*Tool for Genetic Algorithms Implementation*). Esta é capaz de gerar uma nova implementação de Algoritmo Genético, ou pode modificar um Algoritmo Genético (criado pela ferramenta), alterando as etapas independentes da estrutura cromossômica. Nas próximas subseções são apresentados detalhes destas duas formas de utilização e será descrito o funcionamento da ferramenta TOGAI.

Gerar Nova Implementação

O código gerado pela ferramenta TOGAI é dividido em duas grandes partes. A primeira é composta pelas declarações do algoritmo e pelas funções que não serão substituídas pela ferramenta em modificações futuras. A segunda é composta pelas funções que, após escolhidas pelo usuário, serão inseridas automaticamente pela ferramenta e poderão ainda, vir a ser substituídas futuramente. Para percepção de cada parte do código, pela ferramenta, foi criado um separador identificado por */*END SECTION*/*.

A primeira parte do código, chamada de bloco principal, deverá ser modificada pelo usuário visando a adequação dos valores das constantes do AG e a complementação dos conteúdos das funções. Todas as funções pertencentes ao bloco principal são dependentes da estrutura de cromossomo utilizada, e necessitarão de complementação. A estrutura deste bloco está organizada da seguinte forma: Comentário Sobre o Bloco, Inclusão das Bibliotecas, Definição de Constantes, Declaração da Estrutura do Cromossomo, Declaração da função objetivo, Declaração da Estrutura do Indivíduo, Declaração da Estrutura da População, Protótipos das Funções, Funções Dependentes do Cromossomo e as Funções Complementares (Subseção 4.2.2).

Este código está definido em um arquivo de nome *mainCode* e é copiado, pela ferramenta, para um arquivo definido pelo usuário, onde deve ser gerada sua implementação.

Uma vez incluída a primeira parte do código, o passo seguinte da ferramenta é copiar os códigos referentes à segunda parte da implementação, a partir de parâmetros definidos pelo usuário. Esta, chamada de bloco secundário, é composta pelas etapas do AG que são independentes da estrutura cromossômica e possui a seguinte estrutura: Função de Seleção, Função de Evolução da População e Critério de Parada.

Arquivos contendo diferentes implementações das políticas de seleção, evolução da população e critérios de parada ficam armazenados na estrutura de diretório da ferramenta e podem ser utilizados pelo usuário para fazer parte do bloco secundário do seu código.

Para cada uma das fases de seleção, evolução e critério de parada, há um arquivo de configuração onde devem estar definidas todas as políticas disponíveis no sistema. Nestes arquivos podem ainda ser inseridas políticas criadas pelo usuário, que a partir daquele momento farão parte da ferramenta. A Figura 4.13 apresenta, respectivamente, os arquivos de configuração das políticas de seleção, evolução da população e critério de parada. Cada linha deste arquivo (exceto a primeira) contém a identificação de uma política incluída na ferramenta. O formato desta linha deve ser respeitado e segue a seguinte ordem: número de identificação para a política a ser utilizada, descrição textual a ser apresentada na interface gráfica da ferramenta e caminho absoluto do arquivo onde a política está implementada

Arquivo de configuração das políticas de seleção:
1 "Roleta" /TOGAI/selection/Roulette.txt 2 "Torneio" /TOGAI/selection/Tournment.txt 3 "Torneio Proporcional" /TOGAI/selection/PropTourn.txt 4 "Seleção Aleatória" /TOGAI/selection/SUS.txt 5 "Seleção Determinística" /TOGAI/selection/DetermSamp.txt 6 "Seleção Estocástica por Resto Sem Reposição" /TOGAI/selection/ StocNoRemain.txt 7 "Seleção Estocástica por Resto Com Reposição" /TOGAI/selection/ StocRemain.txt 8 "Seleção Estocástica Sem Reposição" /TOGAI/selection/StocNoRepla.txt 9 "Seleção por Ranking" /TOGAI/selection/Ranking.txt
Arquivo de configuração dos critérios de parada: 1 "Número Máximo de Gerações" /TOGAI/stopped/MaxGen.txt 2 "Melhora Mínima da Solução em um Determinado Número de Gerações" /TOGAI/stopped/ImprovMin.txt 3 "Convergência da População" /TOGAI/stopped/PopConverg.txt 4 "Tempo de Computação" /TOGAI/stopped/CompTime.txt 5 "Valor Alvo" /TOGAI/stopped/TargetValue.txt
Arquivo de configuração das políticas de evolução da população: 1 "SGA" /TOGAI/evolution/SGA.txt 2 "SSGA" /TOGAI/evolution/SSGA.txt 3 "Utilizando o Parâmetro GAP" /TOGAI/evolution/GAP.txt

Figura 4.13: Arquivos de configuração das políticas disponíveis na TOGAI.

Alterar Implementação

Após a ferramenta ter gerado o código da aplicação, esta poderá ter suas políticas de seleção, evolução e critério de parada alteradas de forma automática, visando a melhora do desempenho do algoritmo. A alteração da implementação se dá sempre no bloco secundário do código da aplicação, sendo este substituído na íntegra pela ferramenta. O usuário deve fazer uso da ferramenta visando a reconstrução de todo o bloco secundário de código. Quaisquer alterações feitas neste bloco pelo usuário, se desejado, deverão ser gravadas em outro arquivo para que não sejam perdidas.

O Programa TOGAI

A ferramenta foi implementada em *JAVA* e sua utilização é feita através do programa *TOGAI.java* que se encontra no diretório TOGAI da estrutura de diretório da ferramenta. Dentro do diretório TOGAI se encontram os subdiretórios *selection*, *evolution* e *stopped*. No subdiretório *selection* são localizados os arquivos que possuem implementadas as diferentes políticas para a etapa de seleção, bem como o arquivo de configuração destas. Os subdiretórios *evolution* e *stopped* armazenam os mesmos tipos de arquivos do diretório *selection*, só que para as etapas de evolução da população e critério de parada.

Ao executar o programa *TOGAI.java*, uma interface gráfica é aberta contendo as cai-

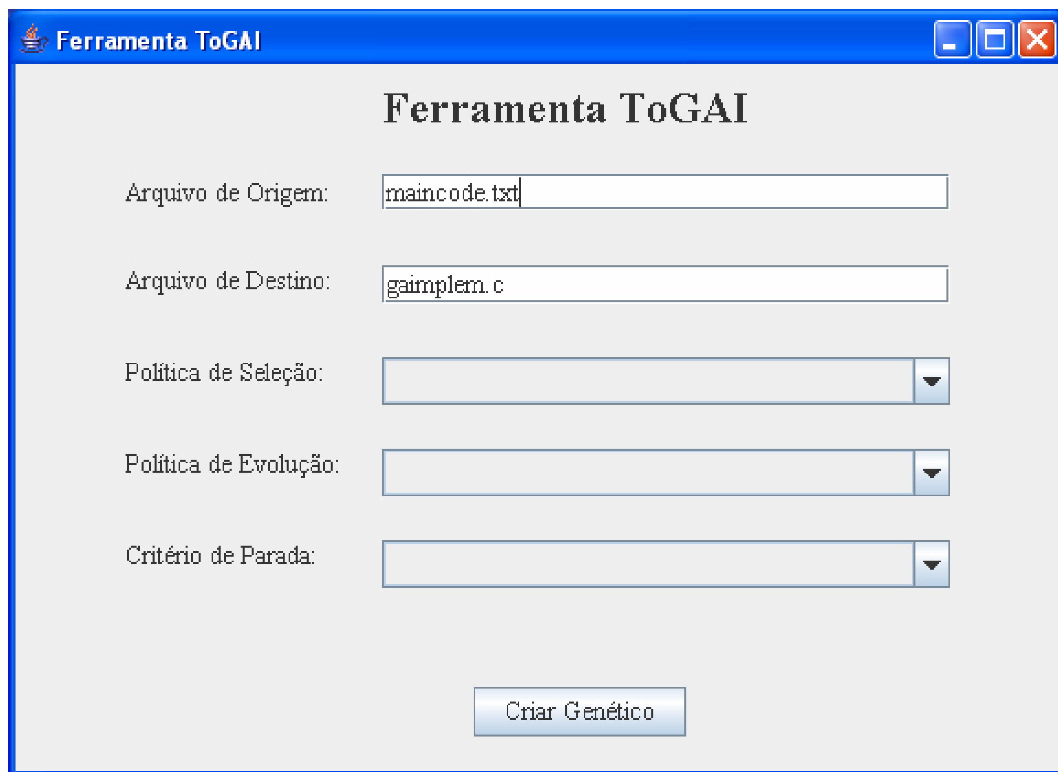


Figura 4.14: Interface da ferramenta TOGAI.

xas de texto e de escolha (*combo boxes*) que serão utilizadas para passar os parâmetros de configuração que serão utilizados para gerar o Algoritmo Genético (Figura 4.14).

A caixa de texto *Arquivo de Origem*, é utilizada para fornecer ao programa o nome do arquivo que terá o seu bloco principal utilizado para compor o AG. O valor padrão para esta caixa de texto é *maincode.txt*, que é a codificação padrão para a estrutura do bloco principal. Caso o usuário já tenha uma implementação gerada, personalizada, e queira reutilizar o bloco principal, o nome do arquivo desta implementação deve ser fornecido nesta caixa de texto. Este reuso visa a alteração de políticas como seleção, evolução e critério de parada que podem ser alteradas e adicionadas ao bloco principal sem alteração do código pelo usuário.

O arquivo de destino deve ser especificado na caixa de texto *Arquivo de Destino*. Este será o nome do arquivo a ser gerado com as políticas definidas pelo usuário que formarão a implementação parcial do AG. É neste arquivo que o usuário deve fazer a complementação de código para o problema específico, finalizando assim, o seu Algoritmo Genético.

Para que seja feita a personalização do AG pela ferramenta, é necessário que sejam escolhidas as políticas de seleção, evolução da população e critério de parada que o usuário deseja utilizar. Estes parâmetros são escolhidos via as caixas de escolha (*combo boxes*) existentes na interface gráfica do programa (Figura 4.14). Cada caixa de escolha apresenta como lista de opções os nomes das políticas cadastradas no arquivo de configuração da etapa do AG.

Por exemplo, para etapa de evolução da população são apresentadas as opções SGA, SSGA e Utilizando o Parâmetro GAP (Figura 4.13).

Uma vez escolhidos os parâmetros de configuração, o botão *Criar Genético* deve ser pressionado para que o modelo de Algoritmo Genético seja gerado. Após a criação do código parcial do AG, pela ferramenta, caberá ao usuário fazer a complementação deste código nas funções que são dependentes da estrutura de dados do cromossomo. Cabe ressaltar que os cabeçalhos destas funções dependentes da estrutura do cromossomo são adicionados ao código para que o usuário preencha seus conteúdos.

Inclusão de Políticas à Ferramenta

Um dos objetivos da ferramenta é possibilitar que o usuário, uma vez tendo criado uma nova política para uma etapa do AG, possa incluí-la na ferramenta para que seja facilmente reutilizada em futuras implementações. Para que isto seja feito, é necessário que o usuário codifique a etapa desejada (seleção de indivíduo, evolução da população ou critério de parada) de acordo com o modelo utilizado (respeitando o cabeçalho definido), copie o arquivo contendo a implementação para dentro do diretório da respectiva etapa e adicione uma entrada no arquivo de configuração da política. A entrada no arquivo de configuração deve conter o número da política no arquivo, uma descrição a ser apresentada na caixa de escolha (*combo box*) da ferramenta e o caminho absoluto do arquivo que contém a implementação (Figura 4.13).

5 Casos de Uso

A fim de validar a ferramenta TOGAI, a mesma foi utilizada para auxiliar na implementação de AGs aplicados a dois problemas distintos encontrados na literatura: o problema da mochila (Knapsack Problem) (PISINGER; MARTELLO; TOTH, 1997) e o problema da filogenia (VIANNA, 2004a).

O problema da mochila foi utilizado com o objetivo de testar a ferramenta e as políticas de seleção disponíveis na mesma. Para este problema, todo o trabalho de implementação dos AGs foi realizado utilizando a ferramenta TOGAI e o modelo de implementação proposto. Nesta implementação, para representação dos cromossomos dos indivíduos da população, foi utilizada a estrutura de dados ”vetor de números binários”. Esta representação é comumente encontrada na literatura e faz parte do Algoritmo Genético simples proposto por Holland (HOLLAND, 1975).

Para testar a ferramenta, com uma representação cromossômica diferente da estrutura de dados ”vetor”, foi usado o problema da filogenia. A codificação de AG utilizada para este problema foi, na verdade, uma reestruturação da implementação feita por Vianna (VIANNA, 2004a) em sua tese de doutorado. Para isto, a codificação do AG de Vianna (VIANNA, 2004a) foi alterada, fazendo com que todos os seus cabeçalhos de funções e regras de implementação fossem adaptados as exigências da ferramenta TOGAI. A representação cromossômica dos indivíduos da população, no AG de Vianna (VIANNA, 2004a), é uma estrutura de dados dinâmica ”árvore binária”. Através desta característica objetivou-se validar a proposta da ferramenta de poder ser utilizada no auxílio a criação de AGs para os mais diversos domínios de problemas.

Na Seção 5.1 o problema da mochila será devidamente abordado, juntamente com as versões de AGs gerados e os resultados dos testes computacionais.

O problema da filogenia será apresentado na Seção 5.2. Apesar deste problema merecer um nível de explicação mais detalhado para o seu perfeito entendimento, aprofundar os seus conceitos foge ao foco deste trabalho. Ainda assim, será feita uma rápida introdução ao problema, as características dos AGs serão mostradas e os testes realizados serão apresentados. Estudos mais detalhados sobre o problema da filogenia podem ser encontrados em (VIANNA,

2004a).

5.1 Problema da Mochila

O problema da mochila (também conhecido como *Knapsack Problem* ou somente KP) é um problema NP-completo bastante estudado nas áreas de Análise de Algoritmos, Otimização Combinatória, Pesquisa Operacional e Inteligência Artificial. Sua forma clássica pode ser descrita da seguinte maneira: tendo-se uma mochila de capacidade C e um conjunto de n objetos diferentes, onde a cada objeto (i) está associado um peso (p_i) e um ganho (v_i), deve-se encontrar um subconjunto de objetos a transportar de modo que o peso total dos elementos selecionados não exceda a capacidade C da mochila e que o ganho total dos objetos selecionados seja o maior possível.

O Problema da Mochila Binário é, talvez, o mais importante Problema da Mochila e um dos mais estudados problemas de programação discreta ((PISINGER; MARTELLO; TOTH, 1997)). A razão para tal interesse está, basicamente, ligada a três fatores:

a) pode ser visto como um dos simples problema de programação linear inteira;

b) aparece como subproblema em muitos outros problemas mais complexos;

c) pode representar uma gama muito grande de situações práticas.

No Problema da Mochila Binário temos a situação em que um único exemplar de cada ítem pode ser selecionado. O modelo matemático do problema é apresentado a seguir:

$$x_i = \begin{cases} 1 & \text{se o item } i \text{ for selecionado;} \\ 0 & \text{caso contrário.} \end{cases}$$

$$\text{onde: } i = 1, ..., n$$

$$\text{Maximizar } \Theta = \textstyle\sum_{i=1}^{n} v_i x_i$$

$$\text{Sujeito a } \textstyle\sum_{i=1}^{n} p_i x_i \leq C$$

$$x_i = 0 \text{ ou } 1,\ i = 1, ..., n.$$

5.1.1 Implementação do Problema da Mochila Binária utilizando Algoritmos Genéticos

No presente trabalho são apresentados Algoritmos Genéticos com o intuito de encontrar a melhor solução para o problema da Mochila Binária. O foco deste estudo é investigar, dentre as políticas de seleção Roleta, Torneio, Torneio Estocástico, Seleção Aleatória, Seleção Determinística, Seleção Estocástica por Resto Sem Reposição, Seleção Estocástica por Resto Com Reposição, Seleção Estocástica Sem Reposição e Seleção por *Ranking* (apresentadas na Seção 2.7), qual apresenta o melhor resultado para o problema.

Para cada versão de AG gerado foi incluída uma política de seleção diferente disponibilizada pela ferramenta. As demais políticas e parâmetros utilizados foram os mesmos para todas as versões e são apresentados nas próximas seções.

População Inicial

O tamanho da população utilizado neste trabalho foi de 100 indivíduos, o que possibilitou uma boa cobertura do espaço de busca do problema e também se mostrou eficiente na prática, pelo recurso computacional disponível.

Os cromossomos são formados por n genes, onde n representa a quantidade de ítens possíveis a fazer parte da mochila e tem o seu valor definido pela instância a ser utilizada. Cada gene pode assumir valores 0 ou 1, indicando, respectivamente, a ausência ou presença do ítem na mochila. Estes ítens são compostos por dois valores, ganho e peso, que serão lidos de um arquivo de texto. Cada indivíduo representa uma solução que será formada por um subconjunto de ítens selecionados para estar na mochila. Para formar indivíduos factíveis para a população inicial foram utilizados quatro critérios: maior ganho, menor peso, maior relação ganho por peso (ganho/peso) e seleção de ítem entre os α maiores ganhos.

O critério maior ganho foi utilizado para gerar o primeiro indivíduo da população. De acordo com este, uma busca é feita para se encontrar os ítens cujos ganhos sejam os maiores entre todos. Para formar o segundo indivíduo, foi utilizado o critério menor peso onde uma varredura é realizada em busca dos ítens cujos pesos sejam os menores. O terceiro indivíduo é formado através do critério maior relação ganho por peso. Assim, para cada ítem é calculada a divisão do ganho pelo peso a fim de se encontrar os maiores resultados. A partir do quarto indivíduo, o critério utilizado foi seleção de ítem entre os α=6 maiores ganhos. Desta forma, a cada varredura, um ítem será selecionado aleatoriamente entre os seis que representam os maiores ganhos. O valor α=6 foi escolhido porque, nos testes com um valor inferior, a população mostrou-se bastante homogênea.

Após a seleção de um ítem, em cada critério, seu peso é verificado para saber se este excede a capacidade da mochila. Se o peso exceder esta capacidade, o ítem é desconsiderado e novamente é aplicado o critério para os ítens remanescentes, senão, atualiza-se a capacidade da mochila e nesta inclui-se o ítem.

Função de Aptidão (*Fitness*)

Neste estudo, o *fitness* é igual ao valor da função objetivo do indivíduo da população. Este valor é calculado somando-se os ganhos dos ítens que compõe o indivíduo. O melhor indivíduo da população será aquele que possuir o maior valor de função objetivo.

Seleção

Foco deste estudo, o processo de seleção é responsável pela escolha de indivíduos sobre os quais serão aplicados os operadores genéticos cruzamento e mutação. Todas as políticas disponibilizadas na ferramenta TOGAI serão utilizadas, sendo que cada uma comporá uma versão de Algoritmo Genético onde as instâncias de teste serão submetidas, visando encontrar as políticas que gerem as melhores soluções para o problema.

Cruzamento

Neste estudo, o cruzamento foi realizado com dois indivíduos selecionados (pais). A política de cruzamento adotada foi *um ponto de corte* (Figura 2.7) que, segundo Vianna (VIANNA, 2004b), é uma política bastante utilizada na literatura. Neste tipo de cruzamento, é definido como ponto de corte $\lfloor n/2 \rfloor$, sendo n o tamanho do indivíduo. Assim, dois indivíduos (filhos) são gerados, sendo a primeira parte do primeiro pai trocada com a primeira parte do segundo pai. A probabilidade de cruzamento foi definida em 80%. Caso não ocorra, os pais serão mantidos na nova geração e, assim como os filhos, poderão ou não sofrer mutação.

Mutação

Da mesma forma que o cruzamento, este operador genético pode ou não ocorrer de acordo com uma dada probabilidade, que neste trabalho foi definida em 10% sobre o indivíduo. Na mutação, os filhos gerados pelo cruzamento, ou os pais que foram mantidos, poderão ter suas características modificadas da seguinte maneira: dados os n genes do indivíduo, um é escolhido randomicamente e seu valor é alterado para 0 se este for 1 e para 1 se este for 0, como pode ser visto na Figura 2.10.

Evolução da População

O método de evolução da população utilizado neste estudo foi o SGA (descrito na Seção 2.10.1), onde, com um tamanho de população fixo, a nova geração substitui a anterior dada às probabilidades de aplicação dos operadores genéticos (cruzamento e mutação) sobre os dois indivíduos escolhidos por um determinado método de seleção.

Validação da Solução

Após a aplicação dos operadores, cada indivíduo terá o somatório de seus pesos verificado. Se este somatório extrapolar a capacidade da mochila, os ítens com as menores relações ganho/peso serão descartados gradativamente até que o somatório dos pesos seja menor ou igual a esta capacidade. Caso este somatório não alcance a capacidade da mochila, com o objetivo de melhorar o indivíduo menos evoluído, tenta-se incluir gradativamente os ítens com as maiores relações ganho/peso verificando se os seus pesos não excedem a capacidade da mochila. Encerrado este processo, para cada indivíduo é calculado o valor de sua função objetivo.

Critério de Parada

O critério de parada escolhido neste caso foi o número máximo de gerações (descrito na Seção 2.11.1). Neste tipo de critério, quando o Algoritmo Genético alcança uma determinada geração predefinida, ele é finalizado, apontando como solução o melhor indivíduo até aquele momento. De acordo com (GOLDBERG, 1989) e (VASCONCELOS et al., 1997), o número ideal de gerações deverá ser definido após a execução de testes para detectar a convergência do algoritmo. Durante os testes, a convergência da melhor solução para a menor instância foi encontrado em média na geração 20. Já para a maior instância, percebeu-se que um número maior do que 1000 gerações poderia melhorar o valor da função objetivo, porém inviabilizando o tempo computacional dos testes. Deste modo, o número de gerações escolhido foi 1000 por não demandar tempo significativo de computação para instâncias pequenas e não demandar tempo proibitivo para as maiores instâncias no equipamento utilizado.

5.1.2 Testes Computacionais

Os testes apresentados nesta seção têm por objetivo avaliar os resultados gerados pelos Algoritmos Genéticos criados para solucionar o problema da mochila binária. Foram utilizadas instâncias geradas por um software desenvolvido por Pisinger ((PISINGER, 2005), (PISINGER, 1999)), que trabalha com os parâmetros n, R e t, onde n representa a quantidade de ítens

que podem fazer parte da mochila, R um fator para definir a faixa de coeficientes e t o tipo de critério utilizado. Para a realização destes testes, de acordo com Martello (MARTELLO; TOTH, 1990), n assumiu os valores 50, 100, 200, 1000, 5000 e 10000. Considerou-se também, segundo Pisinger (PISINGER, 2005), a faixa de coeficientes igual 1000 e o tipo de critério Weakly Correlated Instances (Instâncias Fracamente Correlatas), onde o peso (p_j) de cada ítem j é escolhido randomicamente em [1,R] e o ganho (v_j) em [p_j – R/10, p_j + R/10], sendo v_j maior ou igual a 1. Apesar do nome fracamente correlato, as instâncias têm uma correlação muito alta entre o ganho e o peso de um ítem, diferindo somente por uma pequena porcentagem. Este critério foi escolhido por ser o que mais se aproxima do mundo real (PISINGER, 2005). Para cada valor assumido por n, uma instância foi gerada e um valor para a capacidade da mochila foi calculado a partir da seguinte fórmula: $C = 0.5\sum_{j=1}^{n} p_j$

Onde:

C= capacidade da mochila;

n= quantidade de ítens;

j= índice do ítem;

p_i= peso do ítem i.

Proposta por De Jong (JONG, 1975), a medida de desempenho utilizada nos testes foi a *off-line*. Esta tem por característica levar em conta apenas o valor da melhor função objetivo encontrada no final da execução do algoritmo.

O computador utilizado para a realização dos testes possui processador Intel Pentium IV com 1.70 GHz de freqüência, 512MB de memória principal e sistema operacional Windows XP. Cada algoritmo foi executado 10 vezes para uma mesma instância.

Neste estudo, os resultados dos testes foram avaliados de duas maneiras. Na primeira, o melhor resultado obtido após 10 execuções de um algoritmo para uma determinada instância foi avaliado. Enquanto na segunda, foi considerada a média dos resultados obtidos após as 10 execuções.

Avaliação dos resultados utilizando o maior valor dentre 10 execuções

Serão apresentados a seguir os resultados obtidos com os melhores valores de aptidão para cada política em face das instâncias geradas (vide Figura 5.1). As políticas de seleção utilizadas foram: Roleta, Torneio, Torneio Estocástico, Seleção Aleatória, Seleção Determinística, Seleção Estocástica por Resto Sem Reposição (EstRestoSemRep), Seleção Estocástica por Resto Com Reposição (EstRestoComRep), Seleção Estocástica sem Reposição (EstSemRep)

e Seleção por Ranking.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
n = 50	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00
n = 100	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00
n = 200	54659.00	54657.00	54659.00	54657.00	54657.00	54657.00	54657.00	54661.00	54661.00
n = 1000	273554.00	273556.00	273556.00	273556.00	273554.00	273554.00	273549.00	273560.00	273547.00
n = 5000	1375300.00	1375324.00	1375145.00	1375138.00	1375223.00	1375189.00	1375137.00	1375153.00	1375132.00
n = 10000	2748151.00	2746735.00	2747130.00	2745854.00	2746447.00	2745433.00	2748119.00	2748119.00	2748119.00

Figura 5.1: Melhores resultados das políticas de seleção para 10 execuções com cada instância.

Com o intuito de identificar a política de seleção que apresentou a melhor solução para o Problema da Mochila Binária, a seguinte metodologia foi aplicada: para cada instância, um *ranking* foi feito com o intuito de avaliar o comportamento de cada algoritmo. Em seguida, foi calculada, para cada algoritmo, a média aritmética das colocações obtidas. Desse modo, a melhor política de seleção será aquela que possuir a menor média, conforme Figura 5.2.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
n = 50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
n = 100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
n = 200	6.00	1.00	6.00	1.00	1.00	1.00	1.00	8.00	8.00
n = 1000	3.00	6.00	6.00	6.00	3.00	3.00	2.00	9.00	1.00
n = 5000	8.00	9.00	4.00	3.00	7.00	6.00	2.00	5.00	1.00
n = 10000	9.00	4.00	5.00	2.00	3.00	1.00	6.00	6.00	6.00
Média	4.67	3.67	3.83	2.33	2.67	2.17	2.17	5.00	3.00
Ranking	2.00	4.00	3.00	7.00	6.00	8.00	8.00	1.00	5.00

Figura 5.2: Ranking das políticas de seleção em face dos melhores resultados.

Ao final dos testes, pôde ser constatado que a política de seleção Estocástica por Resto com Reposição foi a que apresentou o melhor valor para função objetivo, ou seja, a melhor solução para o problema (vide Figura 5.3).

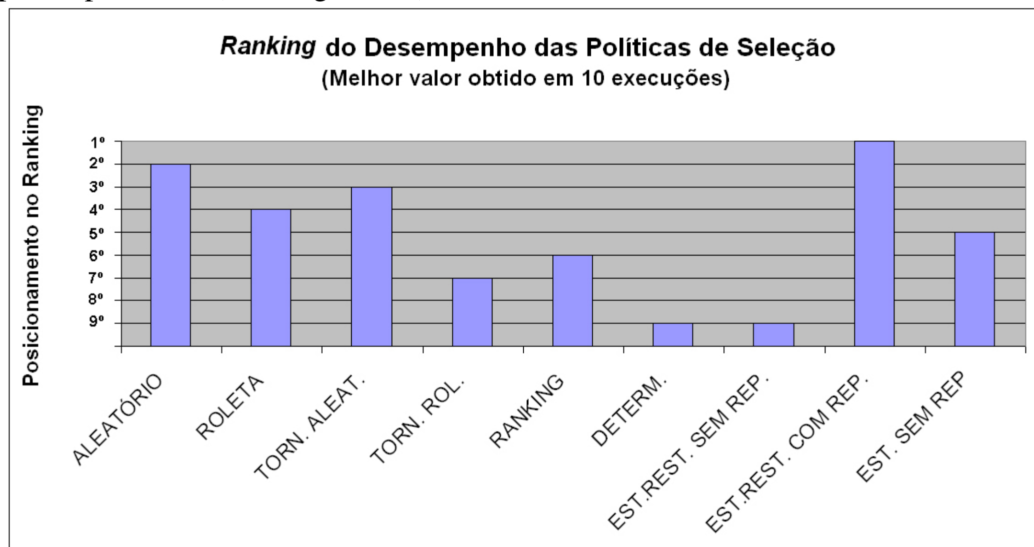


Figura 5.3: Gráfico comparativo do *ranking* das políticas de seleção em face dos melhores resultados.

Por outo lado, as políticas de seleção Determinística e Estocástica por Resto com Reposição foram as que apresentaram os piores valores (vide Figura 5.3).

Avaliação dos resultados utilizando o valor médio dentre 10 execuções

São apresentados a seguir os resultados dos testes obtidos, após as 10 execuções dos algoritmos para cada instância de teste, utilizando a média dos valores de aptidão gerados (vide Figura 5.4). As políticas de seleção utilizadas foram as mesmas usadas nos testes com maior valor, apresentados anteriormente.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
n = 50	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00	13380.00
n = 100	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00	29128.00
n = 200	54653.60	54654.50	54654.40	54652.20	54653.40	54651.90	54654.60	54653.80	54656.60
n = 1000	273545.60	273543.10	273543.10	273543.50	273546.10	273543.60	273541.80	273546.80	273544.00
n = 5000	1375166.70	1375100.80	1375026.50	1375078.60	1375105.20	1375001.60	1375073.00	1375098.30	1374935.10
n = 10000	2748127.00	2746488.50	2745800.70	2745476.00	2746390.60	2745304.80	2745911.90	2746986.30	2746331.00

Figura 5.4: Resultados médios das políticas de seleção para 10 execuções com cada instância.

A política de seleção que apresentou a melhor solução para o Problema da Mochila Binária, foi a mesma do teste anterior. Para cada instância, um *ranking* foi feito com o intuito de avaliar o comportamento de cada algoritmo. Em seguida, foi calculada, para cada algoritmo, a média aritmética das colocações obtidas. Desse modo, a melhor política de seleção será aquela que possuir a menor média, conforme Figura 5.5.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
n = 50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
n = 100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
n = 200	4.00	7.00	6.00	2.00	3.00	1.00	8.00	5.00	9.00
n = 1000	7.00	2.00	2.00	4.00	8.00	5.00	1.00	9.00	6.00
n = 5000	9.00	7.00	3.00	5.00	8.00	2.00	4.00	6.00	1.00
n = 10000	9.00	7.00	3.00	2.00	6.00	1.00	4.00	8.00	5.00
Média	5.17	4.17	2.67	2.50	4.50	1.83	3.17	5.00	3.83
Ranking	1.00	4.00	7.00	8.00	3.00	9.00	6.00	2.00	5.00

Figura 5.5: *Ranking* das políticas de seleção em face dos resultados médios.

Ao final dos testes, pôde ser constatado que a política de seleção Aleatória foi a que apresentou o melhor valor para função objetivo, ou seja, a melhor solução para o problema. Enquanto que a política de seleção Determinística foi a que apresentou o pior valor (vide Figura 5.6).

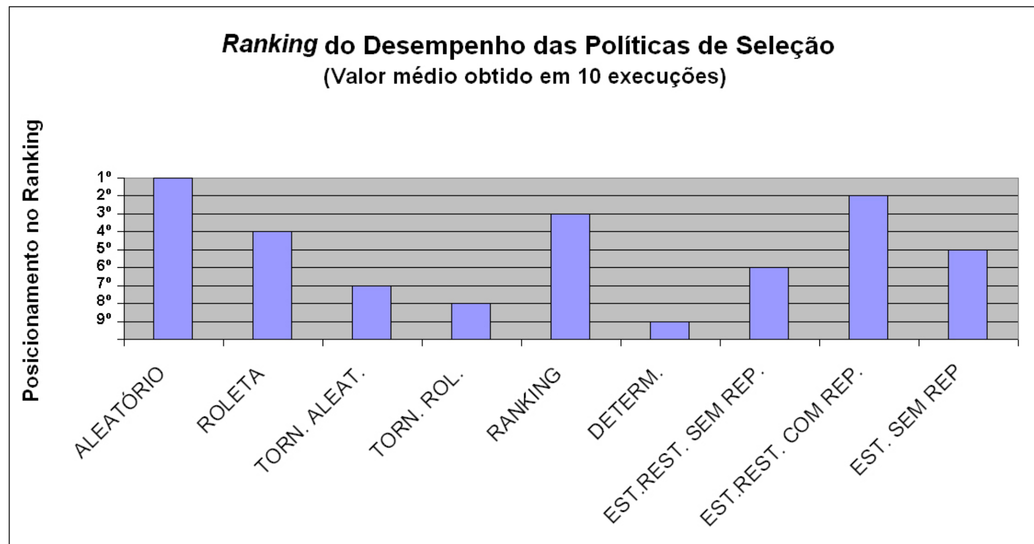


Figura 5.6: Gráfico comparativo do *ranking* das políticas de seleção em face dos resultados médios.

5.1.3 Conclusão do Caso de Uso

Após comparar as políticas de seleção, foi constatado que estas não geraram resultados diferentes para a função objetivo, quando aplicadas a instâncias de cromossomos pequenos com até 100 genes. Para as demais instâncias, foram analisados os maiores valores e os valores médios gerados nas execuções.

Como pode ser observado, a política de seleção Estocástica por Resto com Reposição e a Aleatória foram as que conseguiram os melhores resultados nos testes.

Numa primeira análise, a política Aleatória poderia ser considerada uma má escolha devido ao seu funcionamento não levar em consideração a adaptação do indivíduo. Esta característica possibilita que indivíduos que representam soluções ruins sejam escolhidos com a mesma probabilidade dos bons indivíduos. Contudo, a política aleatória gera diversidade na população, aumentando a cobertura do espaço de busca do problema e possibilitando sair mais facilmente de soluções locais em direção a solução global. Este fato, provavelmente, foi o mais relevante para o bom desempenho desta política em face das instâncias de testes geradas. Com isto, fica constatado mais uma vez que, a melhor escolha por um parâmetro ou política de funcionamento do Algoritmo Genético é na grande maioria das vezes feita através de testes empíricos.

5.2 Problema da Filogenia

Um dos problemas centrais em biologia comparativa (sistemática) é o esclarecimento das relações de ancestralidade entre espécies, grupos de espécies, populações de espécies distin-

tas, populações de uma mesma espécie ou genes homólogos em populações de espécies distintas (mas próximas do ponto de vista evolutivo) (AYALA, 1995; SWOFFORD; OLSEN, 1990). Estas entidades podem ser classificadas sob a designação comum de taxon. A expressão da ancestralidade é feita através de uma árvore com raiz, onde as folhas representam os taxons sob análise e os nós internos representam ancestrais hipotéticos. Diz-se que esta árvore é uma árvore filogenética, uma árvore evolucionária ou uma *filogenia*. Os taxons sob análise são ditos *taxons operacionais*, enquanto os taxons eventualmente associados aos nós internos da árvore filogenética são denominados *taxons hipotéticos*. A associação de um taxon hipotético a cada nó interno de uma filogenia é uma *reconstrução*.

Cada taxon possui um conjunto de atributos denominados características. Uma característica pode representar um atributo morfológico (por exemplo, o tipo de mandíbula de espécies de peixes) ou uma posição de aminoácidos ou nucleotídeos numa proteína (por exemplo, proteínas como as hemoglobinas e os fibrinospeptídeos). A suposição fundamental sobre o conjunto de características consideradas na análise é que sejam independentes entre si. Isto simplifica a análise, pois não há necessidade de modelar o problema considerando a correlação entre as características.

Uma característica possui um conjunto discreto e finito de estados que podem ser assumidos por cada taxon. Este conjunto de estados é o domínio ou conjunto subjacente da característica.

A avaliação da qualidade de uma árvore filogenética depende do critério de otimização empregado para inferi-la. O critério pode ser baseado num modelo estocástico, na compatibilidade apresentada pelos dados, na comparação de distâncias num espaço métrico ou no *princípio de parcimônia*, sendo os dois últimos os mais comumente utilizados. No primeiro critério, as características são formuladas em um espaço métrico. Um espaço métrico é um par (C, d), onde C é o conjunto subjacente e d:$C \times C \rightarrow \mathbb{R}$ é uma função de distância que tem as seguintes propriedades:

- $d(x,y) \geq 0, \forall x,y \in C$;
- $d(x,y) = 0 \Leftrightarrow x = y, \forall x,y \in C$;
- $d(x,y) = d(y,x), \forall x,y \in C$;
- $d(x,y) \leq d(x,z) + d(z,y), \forall x,y,z \in C$.

O grafo de transição de estados de uma característica que representa um espaço métrico é não direcionado e sua matriz de adjacência é simétrica.

Este critério é bastante utilizado em trabalhos sobre genoma (ARAÚJO; ALMEIDA, 2002; GALLUT; BARRIEL; VIGNES, 2000; WANG; WARNOW, 2001). Por exemplo, seja T uma árvore na qual os nós são representados por genomas. Seja G_i e G_j dois nós de T. Seja P_{ij} o caminho entre G_i e G_j em T, e seja k_e o número real de eventos ao longo da aresta $e \in P_{ij}$. A *distância evolucionária* entre G_i e G_j é $k_{ij} = \Sigma_{e \in P_{ij}} k_e$.

Pelo critério de parcimônia (SWOFFORD; OLSEN, 1990), a melhor árvore filogenética (mais parcimoniosa) é aquela que pode ser explicada pelo menor número de *passos evolutivos* (EDWARDS; CAVALLI-SFORZA, 1964; HENNIG, 1966). É frequentemente afirmado que o critério de parcimônia pode ser legitimado como o melhor critério de otimização na obtenção de árvores filogenéticas, desde que a probabilidade de ocorrerem mudanças evolucionárias seja pequena (PENNY; FOULDS; HENDY, 1982; SOBER, 1987). O critério de parcimônia é utilizado ao longo deste trabalho.

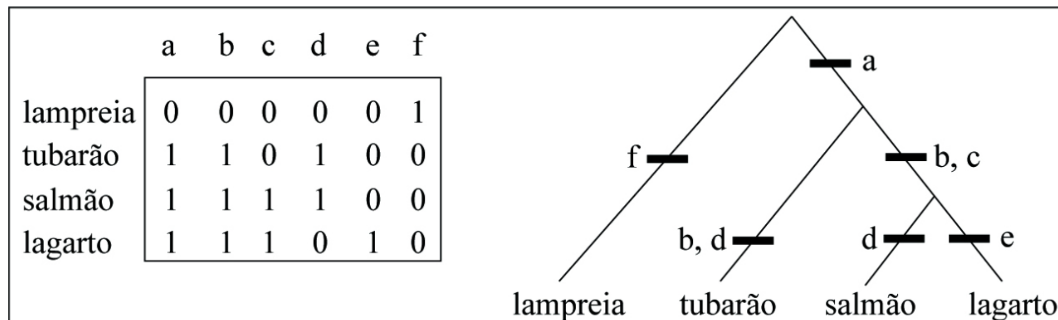


Figura 5.7: Exemplo de um conjunto de taxons e uma filogenia associada.

A Figura 5.7 apresenta um conjunto de taxons e uma filogenia para este (ambos adaptados de (KITCHING et al., 1998)). Os dados são binários e estão representados por uma matriz, onde as linhas são indexadas por taxons e as colunas pelas características. As características presentes são: (a) pares de nadadeiras, (b) mandíbulas, (c) grandes ossos dérmicos, (d) barbatanas em fila, (e) pulmões, (f) línguas ásperas. O valor atribuído ao par (i, j) é 1 se o taxon i possui a característica j e 0 se a característica estiver ausente. A filogenia mostrada assume que os taxons descendem de um ancestral comum (raiz da árvore) onde todas as características estão ausentes. São sinalizados na figura os pontos onde cada característica muda de estado.

Aplicações

A inferência de árvores filogenéticas é importante para a fundamentação de taxonomias, que por sua vez tem uma estreita relação com a paleontologia (WILSON, 1992). Isso pode ocorrer de duas maneiras:

1. Espécimes fósseis podem apresentar estados para características morfológicas que não

estão presentes nas espécies contemporâneas, podendo indicar também uma ordenação provável para os conjuntos de estados das características.

2. A introdução de uma espécie já extinta na análise após a obtenção de uma árvore evolucionária é uma das formas de testar sua plausibilidade.

Em biogeografia, árvores evolutivas auxiliam na identificação de uma possível correlação entre mudanças ambientais e mudanças morfológicas sobre uma mesma espécie (co-evolução ambiente-espécie) (WILEY et al., 1991). Em biologia molecular, dois genes são *homólogos* se possuem um ancestral comum. Os taxons operacionais podem ser genes homólogos de uma espécie ou de espécies distintas. A comparação entre os genes permite inferir tamanhos de populações de uma espécie ao longo do tempo (AYALA, 1995). Esta inferência é conseguida pela comparação das variações moleculares inter/extra-espécies de genes homólogos. Utilizando-se um modelo probabilístico para as variações de nucleotídeos nas seqüências e sabendo-se quando a função genética passou a ser realizada por organismos vivos, é possível estimar estatisticamente o tamanho populacional mínimo para gerar um subconjunto dos genes que foi agrupado monofileticamente numa árvore evolucionária proposta.

Há algum tempo atrás, utilizou-se a análise filogenética como evidência numa acusação de tentativa de homicídio (VOGEL, 1997). O acusado, médico de um grupo de infectados pelo vírus HIV, teria injetado sangue contaminando numa mulher. Comparou-se a população virótica da mulher com as populações viróticas dos pacientes do acusado e de uma amostragem das populações do vírus na América. O resultado apontou que o vírus que infectou a vítima era filogeneticamente muito mais próximo da população virótica dos pacientes do acusado do que do restante das populações utilizadas na análise.

5.2.1 Implementação do Problema da Filogenia utilizando AGs

Como comentado anteriormente, a implementação de Algoritmo Genético para este caso de uso é uma reestruturação do código apresentado por Vianna (VIANNA, 2004a) em sua tese de doutorado. A reestruturação do código de Vianna (VIANNA, 2004a) incluiu, além das adaptações das funções às regras da ferramenta, a alteração da etapa de evolução da população e a alteração da etapa de seleção de indivíduos. Esta última foi, do mesmo modo que no problema da mochila, alterada para diferentes políticas com o objetivo de investigar a melhor para o problema abordado.

Para cada versão de AG gerado foi incluída uma das políticas de seleção disponibilizadas pela ferramenta. As demais políticas e parâmetros utilizados foram os mesmos para todas as versões e são apresentados nas próximas seções.

População Inicial

Na fase inicial do Algoritmo Genético, uma população de 100 indivíduos foi gerada através de um algoritmo de construção randomizado. Usou-se nesta etapa o algoritmo *GStep* (ANDREATTA, 1998) (ANDREATTA; RIBEIRO, 2002) para a construção de cada indivíduo da população, pois os estudos realizados em (ANDREATTA, 1998) (ANDREATTA; RIBEIRO, 2002) mostraram que este encontra, geralmente, as melhores soluções (embora o esforço computacional exigido seja mais elevado quando comparado com outros algoritmos).

Função de Aptidão (*Fitness*)

Como dito anteriormente, o critério de avaliação da qualidade de uma árvore filogenética utilizado neste trabalho é baseado no princípio da parcimônia (SWOFFORD; OLSEN, 1990). Este critério assume que a melhor filogenia é aquela que possui o número mínimo de passos evolutivos (mudança no estado de uma característica) (EDWARDS; CAVALLI-SFORZA, 1964; HENNIG, 1966). Em outras palavras, dada uma filogenia s, a tarefa de avaliá-la consiste em definir o conjunto de características (taxons hipotéticos) de cada nó interno da árvore de tal forma que o número de mudanças de estado seja o menor possível.

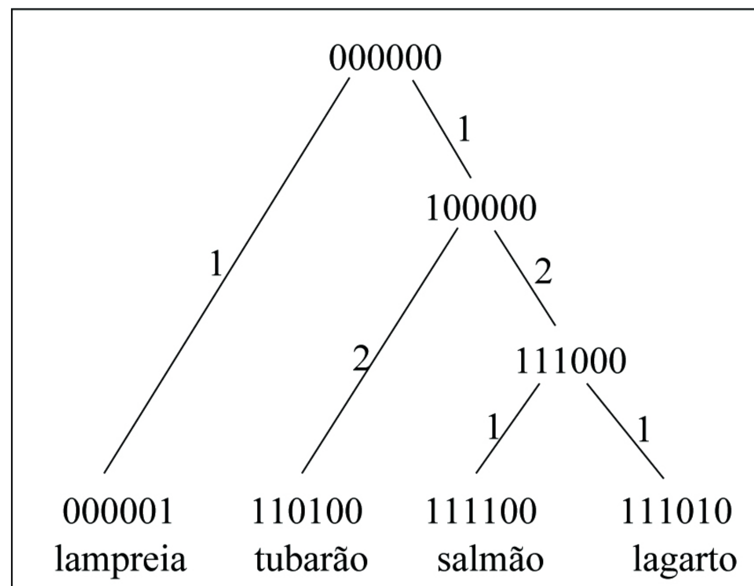


Figura 5.8: Exemplo de uma filogenia bem avaliada.

A Figura 5.8 mostra uma filogenia que possui um conjunto de taxons hipotéticos baseados na evolução descrita na Figura 5.7. Nesta árvore, em cada aresta é sinalizada a quantidade de passos evolutivos ocorridos, totalizando um total de oito em toda a filogenia. Já na Figura 5.9 houve uma melhor definição dos taxons hipotéticos dos nós internos, obtendo-se um número de passos evolutivos igual a sete, que representa o menor valor de parcimônia para tal filogenia.

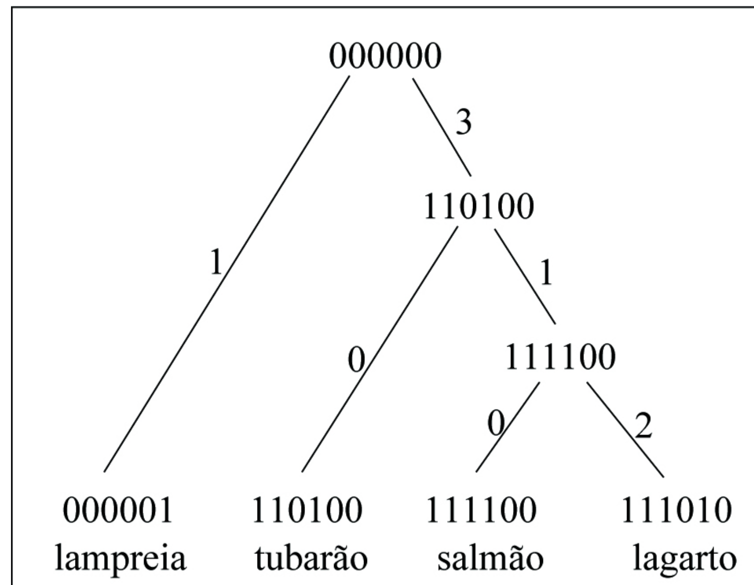


Figura 5.9: Exemplo de uma filogenia bem avaliada.

Maiores detalhes a respeito do algoritmo utilizado para cálculo do número de passos evolutivos, podem ser encontrados em (VIANNA, 2004a).

Seleção

Na implementação de Algoritmo Genético criada por Vianna (VIANNA, 2004a) foi adotado o método da roleta (descrito na Seção 2.7.1) como política de seleção de indivíduos. O foco deste caso de uso é substituir o método da roleta pelos demais métodos de seleção disponibilizados pela ferramenta com o objetivo de investigar se a escolha feita por Vianna (VIANNA, 2004a) realmente foi a melhor para o problema abordado.

Cruzamento

A política de cruzamento utilizada neste trabalho foi a reconexão por caminhos. A técnica de reconexão por caminhos é uma estratégia de intensificação originalmente proposta por Glover (GLOVER, 1994) para explorar trajetórias entre soluções elites obtidas pela busca tabu ou pela heurística *scatter search* (busca dispersa). O uso desta técnica, como extensão da operação de cruzamento tradicional, foi uma contribuição original da tese de Vianna (VIANNA, 2004a). Ao invés de produzir apenas um filho, este investiga um grupo de soluções compartilhando as características dos pais. A solução encontrada pela reconexão por caminhos é o melhor filho que pode ser obtido por uma operação de cruzamento convencional.

Sejam *s1* e *s2*, respectivamente, a filogenia inicial e a filogenia guia para a reconexão por

caminhos. Sejam *N1* um nó de *s1* , e *N2* o seu respectivo em s2 . O objetivo da reconexão por caminhos é fazer com que, após um certo número de iterações, o conjunto de taxons operacionais existentes na subárvore filha esquerda de *N1* seja igual ao conjunto de taxons operacionais existentes na subárvore esquerda de *N2*.Consequentemente, isso faz com que o conjunto de taxons operacionais na subárvore direita de *N1* seja igual ao conjunto da subárvore direita de *N2*. Esse processo é iniciado na raiz de *s1* e é propagado até suas folhas.

Mutação

A etapa de mutação do indivíduo, para este problema, se diferencia daquela apresentada na Seção 2.9, principalmente devido a representação cromossômica do indivíduo. Como esta representação é uma estrutura de dados dinâmica ”árvore binária”, a mutação é realizada através do reposicionamento de um ramo (subárvore) da árvore. Inicialmente um nó é escolhido aleatoriamente para que a subárvore ligada a este seja desconectada. Em seguida um novo nó é escolhido aleatoriamente para que a subárvore seja novamente reconectada.

Evolução da População

Na implementação original desenvolvida por Vianna (VIANNA, 2004a), para a evolução da população seguiu-se a estratégia proposta em (BEAN, 1994). Nela, a evolução da população ao longo das gerações é realizada da seguinte forma:

A cada geração a população é ordenada e dividida em classes. A classe A é composta pelos melhores indivíduos, que representam 30% da população. A classe C pelos piores indivíduos, que representam 20% da população. A classe B é formada pelos indivíduos restantes. Para gerar a população da geração k+1, a partir da geração k, os seguintes passos são executados.

- Todos os indivíduos da classe A são copiados para a próxima geração.
- Todos os elementos da classe C são reconstruídos, ou seja, 20% dos indivíduos de cada geração são construídos utilizando o algoritmo de construção aleatorizado *GStep* (ANDREATTA; RIBEIRO, 2002) (ANDREATTA, 1998). É o mesmo procedimento utilizado na criação da população inicial.
- Os indivíduos restantes são provenientes de um cruzamento entre um representante escolhido aleatóriamente na classe A com um representante também escolhido aleatoriamente nas classes B ou C.

Como esta política de evolução é específica da implementação de Vianna (VIANNA, 2004a), optou-se por fazer a alteração nesta etapa, substituindo a política utilizada por Vianna (VIANNA, 2004a) por outra disponibilizada pela ferramenta. A política utilizada para o problema foi a SGA (descrita na Seção 2.10.1), que desta forma pôde ser validada para o problema da filogenia.

Critério de Parada

O critério de parada utilizado por Vianna (VIANNA, 2004a) na sua implementação foi o número máximo de gerações (descrito na Seção 2.11.1). O número de gerações definido para execução do Algoritmo Genético foi 100.

5.2.2 Testes Computacionais

Os testes apresentados nesta seção têm por objetivo avaliar os resultados gerados pelos Algoritmos Genéticos criados para solucionar o problema da filogenia. Foram geradas variações de AGs contemplando cada uma das políticas de selação disponíveis na ferramenta. Cada um dos AGs foi avaliado com base em oito problemas reais de inferência de filogenias que são apresentados na Figura 5.10. Na figura são apresentados para cada instância o seu nome, o número n de taxons, o número m de características e, na coluna *melhor*, os melhores resultados conhecidos. Sete destes problemas foram obtidos através de contato com os editores da revista *Cladistics* (http://www.cladistics.org/journal.html). Estes problemas foram utilizados por Vianna (VIANNA, 2004a) e têm sido utilizados na comparação de programas para inferência de filogenias (LUCKOW; PIMENTEL, 1985; PLATNICK, 1987; PLATNICK, 1989). Além destas instâncias, foi utilizado também a instância GOLO (GOLOBOFF, 1997) fornecida por Vianna (VIANNA, 2004a).

Instância	n	m	*melhor*
ANGI	49	59	216
GRIS	47	93	172
TENU	56	179	682
ETHE	58	86	372
ROPA	75	82	325
GOLO	77	97	496
SCHU	113	146	759
CARP	117	110	548

Figura 5.10: Melhores resultados para as oito instâncias da literatura.

Da mesma forma que no estudo de caso do problema da mochila, a medida de desempenho utilizada nos testes foi a *off-line*, proposta por De Jong (1975) (JONG, 1975), que leva em cosideração apenas o valor da melhor função objetivo encontrada no final da execução do algoritmo.

O computador utilizado para a realização dos testes possui processador Pentium M com 1.86GHz de freqüência, 1 GB de memória principal e sistema operacional Windows XP. Cada algoritmo foi executado 5 vezes para uma mesma instância.

Neste estudo, os resultados dos testes foram avaliados de duas maneiras. Na primeira, o melhor resultado obtido após 5 execuções de um algoritmo para uma determinada instância foi avaliado. Enquanto na segunda, foi considerada a média dos resultados obtidos após as 5 execuções.

Avaliação dos resultados utilizando o maior valor dentre 5 execuções

Serão apresentados a seguir os resultados obtidos com os melhores valores de aptidão para cada política em face das instâncias utilizadas (vide Figura 5.11). As políticas de seleção utilizadas foram: Roleta, Torneio, Torneio Estocástico, Seleção Aleatória, Seleção Determinística, Seleção Estocástica por Resto Sem Reposição (EstRestoSemRep), Seleção Estocástica por Resto Com Reposição (EstRestoComRep), Seleção Estocástica sem Reposição (EstSemRep) e Seleção por Ranking.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
GRIS	172.00	172.00	172.00	172.00	172.00	172.00	172.00	172.00	172.00
ANGI	216.00	216.00	216.00	217.00	217.00	217.00	217.00	217.00	216.00
TENU	683.00	682.00	682.00	682.00	682.00	683.00	682.00	682.00	682.00
ETHE	372.00	372.00	373.00	373.00	373.00	374.00	372.00	372.00	372.00
GOLO	498.00	501.00	501.00	506.00	503.00	501.00	500.00	501.00	498.00
ROPA	326.00	327.00	327.00	328.00	327.00	326.00	326.00	326.00	328.00
SCHU	763.00	762.00	764.00	765.00	761.00	764.00	759.00	762.00	762.00
CARP	551.00	550.00	554.00	555.00	552.00	553.00	550.00	549.00	551.00

Figura 5.11: Melhores resultados das políticas de seleção para 5 execuções com cada instância.

Com o intuito de identificar a política de seleção que apresentou a melhor solução para o Problema da Filogenia, a seguinte metodologia foi aplicada: para cada instância, um *ranking* foi feito com o intuito de avaliar o comportamento de cada algoritmo. Em seguida, foi calculada, para cada algoritmo, a média aritmética das colocações obtidas. Desse modo, a melhor política de seleção será aquela que possuir a menor média, conforme Figura 5.12.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
GRIS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ANGI	1.00	1.00	1.00	5.00	5.00	5.00	5.00	5.00	1.00
TENU	8.00	1.00	1.00	1.00	1.00	8.00	1.00	1.00	1.00
ETHE	1.00	1.00	6.00	6.00	6.00	9.00	1.00	1.00	1.00
GOLO	1.00	4.00	4.00	9.00	8.00	4.00	3.00	4.00	1.00
ROPA	1.00	6.00	6.00	9.00	6.00	1.00	1.00	1.00	1.00
SCHU	6.00	3.00	7.00	9.00	2.00	7.00	1.00	3.00	3.00
CARP	4.00	2.00	8.00	9.00	6.00	7.00	2.00	1.00	4.00
MÉDIA	2.88	2.38	4.25	6.13	4.38	5.25	1.88	2.13	1.63
RANKING	5.00	4.00	6.00	9.00	7.00	8.00	2.00	3.00	1.00

Figura 5.12: Ranking das políticas de seleção em face dos melhores resultados.

Ao final dos testes, pôde ser constatado que a política de seleção Estocástica sem reposição foi a que apresentou o melhor valor para função objetivo, ou seja, a melhor solução para o problema. Por outro lado, a política de seleção Torneio (variação roleta) foi a que apresentou o pior valor (vide Figura 5.13).

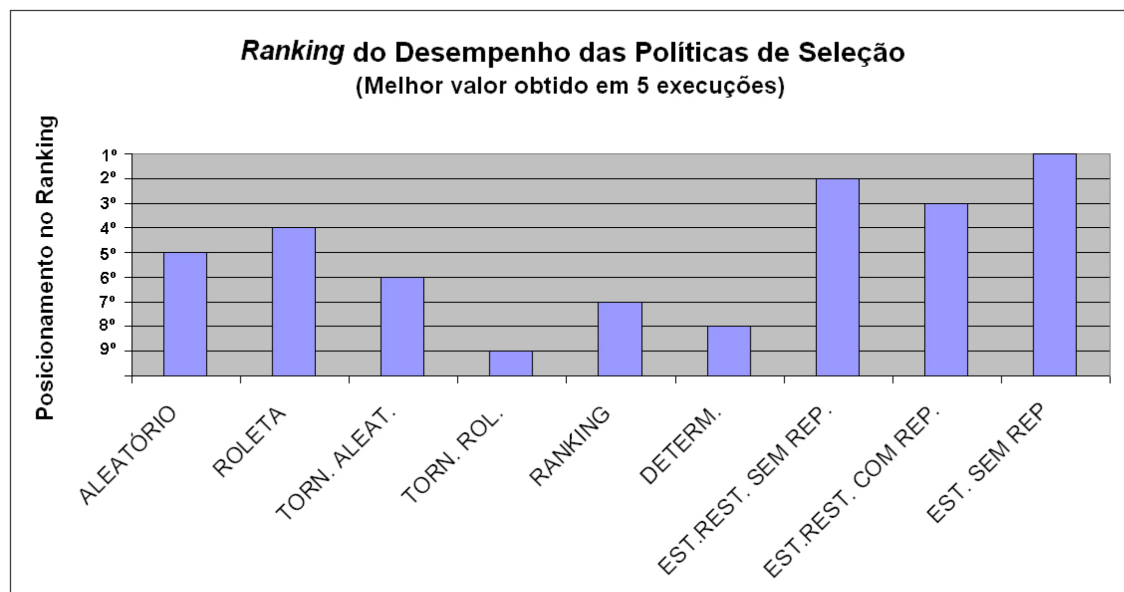


Figura 5.13: Gráfico comparativo do *ranking* das políticas de seleção em face dos melhores resultados.

Avaliação dos resultados utilizando o valor médio dentre 5 execuções

São apresentados a seguir os resultados dos testes obtidos, após as 5 execuções dos algoritmos para cada instância de teste, utilizando a média dos valores de aptidão gerados (vide Figura 5.14). As políticas de seleção utilizadas foram as mesmas utilizadas para os testes com maior valor, apresentados anteriormente.

A metodologia utilizada para identificar a política de seleção melhor adaptada ao pro-

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
GRIS	172.00	172.00	172.20	172.00	172.20	172.20	172.20	172.00	172.00
ANGI	217.00	217.20	217.80	218.80	217.40	217.80	217.20	217.20	216.80
TENU	683.00	682.60	683.60	683.20	683.40	683.00	682.80	682.40	683.00
ETHE	373.60	373.20	374.60	374.60	374.00	374.20	373.00	373.40	373.80
GOLO	499.80	502.80	504.00	508.60	504.00	504.20	502.20	502.40	502.40
ROPA	327.60	328.00	328.00	328.40	328.00	327.40	326.80	327.60	328.00
SCHU	767.80	764.00	768.80	773.20	766.80	770.20	763.00	765.40	764.80
CARP	552.40	552.20	555.80	556.80	555.00	555.20	552.40	551.00	553.00

Figura 5.14: Resultados médios das políticas de seleção para 5 execuções com cada instância.

blema foi a mesma do teste anterior. Para cada instância, um *ranking* foi feito com o intuito de avaliar o comportamento de cada algoritmo. Em seguida, foi calculada, para cada algoritmo, a média aritmética das colocações obtidas. Desse modo, a melhor política de seleção será aquela que possuir a menor média, conforme Figura 5.15.

INSTÂNCIA	ALEATÓRIO	ROLETA	TORN. ALEAT.	TORN. ROL.	RANKING	DETERM.	EST.REST. SEM REP.	EST.REST. COM REP.	EST. SEM REP
GRIS	1.00	1.00	6.00	1.00	6.00	6.00	6.00	1.00	1.00
ANGI	2.00	3.00	7.00	9.00	6.00	7.00	3.00	3.00	1.00
TENU	4.00	2.00	9.00	7.00	8.00	4.00	3.00	1.00	4.00
ETHE	4.00	2.00	8.00	8.00	6.00	7.00	1.00	3.00	5.00
GOLO	1.00	5.00	6.00	9.00	6.00	8.00	2.00	3.00	3.00
ROPA	3.00	5.00	5.00	9.00	5.00	2.00	1.00	3.00	5.00
SCHU	6.00	2.00	7.00	9.00	5.00	8.00	1.00	4.00	3.00
CARP	3.00	2.00	8.00	9.00	6.00	7.00	3.00	1.00	5.00
MÉDIA	3.00	2.75	7.00	7.63	6.00	6.13	2.50	2.38	3.38
RANKING	4.00	3.00	8.00	9.00	6.00	7.00	2.00	1.00	5.00

Figura 5.15: Ranking das políticas de seleção em face dos resultados médios.

Ao final dos testes, pôde ser constatado que a política de seleção Estocástica por Resto com Reposição foi a que apresentou o melhor valor para função objetivo, ou seja, a melhor solução para o problema. Já a política de seleção Torneio (variação roleta) foi a que apresentou o pior valor (vide Figura 5.16).

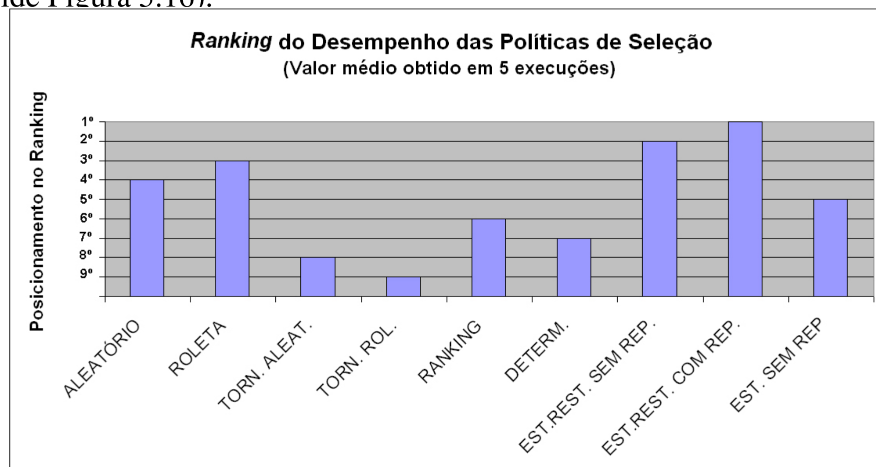


Figura 5.16: Gráfico comparativo do *ranking* das políticas de seleção em face dos resultados médios.

5.2.3 Conclusão do Caso de Uso

Como pode ser observado, a política de seleção Estocástica sem Reposição foi a que obteve o melhor resultado para os testes com melhor valor, enquanto a política de seleção Estocástica por Resto com Reposição foi a que obteve o melhor resultado para teste com valor médio. O pior resultado entre todas as políticas, tanto para os testes utilizando valor máximo quanto para os testes utilizando valor médio, foi obtido pelo método Torneio (variação roleta).

A política Roleta, escolhida por Vianna (VIANNA, 2004a) para fazer parte do seu AG, obteve no teste que utiliza o melhor valor a quarta colocação e no teste que utiliza a média dos valores a terceira colocação. Desta forma pode-se concluir que melhores resultados poderiam ser atingidos caso a política de seleção utilizada por Vianna (VIANNA, 2004a) fosse uma das Estocásticas que se destacaram nos testes.

6 Trabalhos Futuros

A ferramenta TOGAI foi desenvolvida visando seu contínuo crescimento. O modelo de implementação utilizado objetiva facilitar a inclusão de políticas de seleção, evolução da população e critérios de parada à ferramenta. Usuários que necessitem utilizar políticas diferentes daquelas disponibilizadas poderão, de forma facilitada, adicionar e testar novas políticas. Desta forma a ferramenta motiva novos trabalhos de pesquisa e inclusão de políticas para diferentes etapas do AG.

Neste trabalho, para a etapa de evolução da população foram implementadas as políticas SGA, SSGA e GAP. Para as políticas SSGA e GAP é necessário que se escolha qual ou quais indivíduos da geração anterior serão substituídos pelos novos indivíduos. Nesta primeira versão da ferramenta, a única estratégia de substituição de indivíduos implementada foi a substituição dos indivíduos menos adaptados, isto é, a substituição das piores soluções. Estão sendo implementadas novas políticas de substituição de indivíduos que futuramente serão adicionadas à ferramenta.

O desenvolvimento da ferramenta TOGAI foi feito visando solucionar problemas de otimização mono-objetivo, contudo poderá ser feita uma análise do modelo de implementação a fim de viabilizar uma expansão da ferramenta para solucionar problemas multiobjetivo.

Nesta primeira versão da ferramenta, os AGs gerados são seqüenciais, não aproveitando desta forma o paralelismo implícito nestes algoritmos. Com o intuito de aproveitar estas características paralelas, propõe-se uma avaliação da ferramenta TOGAI afim de verificar a possibilidade de adição de modelos de implementação paralela que possam ser utilizados, de forma facilitada pelo usuário, para melhorar o desempenho de sua implementação.

A ferramenta TOGAI possui uma interface gráfica que permite ao usário fazer a escolha dos parâmetros a serem utilizados nos AGs. Porém, uma vez definidos os parâmetros, estes serão mantidos constantes durante todo o processamento do AG. Uma possível contribuição para a ferramenta é a implementação de procedimentos que modifiquem, em tempo de execução, os valores desses parâmetros de acordo com o comportamento do algoritmo. Como exemplo, pode-se ter um AG em que num determinado instante está acontecendo a convergência dos

valores das funções de adaptação dos indivíduos da população, neste momento poderia ser aumentada a probabilidade de mutação visando uma nova diversificação da população.

O AG padrão proposto por Holland trabalha com indivíduos formados por cromossomos binários. Devido a ampla utilização deste modelo de algoritmo, agregaria valor a ferramenta a adição de implementações das etapas dos AGs dependentes da estrutura cromossômica binária, porém independentes do problema. Poderiam ser implementadadas, por exemplo, políticas diferentes para cruzamento, mutação e população inicial, contudo a implementação da função de adaptação é dependente do problema e ainda seria da responsabilidade do usuário a sua codificação.

Diversas políticas e procedimentos implementados na ferramenta TOGAI incluem a geração de números aleatórios. Neste trabalho as funções utilizadas para a geração destes números foram àquelas disponibilizadas pela Linguagem C ANSI. Pesquisas poderão ser feitas no sentido de alterar a função de geração de números aleatórios padrão para outras disponíveis na literatura, buscando desta forma a melhoria da codificação e da precisão dos AGs.

Poderá ser analisada a possibilidade de uso de políticas combinadas para tentar melhorar o desempenho dos AG.

7 Conclusão

Problemas de otimização combinatória aparecem freqüentemente em vários setores da economia. Grande parte deles são intratáveis por natureza ou são grandes o suficiente para tornar inviável o uso de algoritmos exatos. O uso de Algoritmos Genéticos (AGs) é uma das principais estratégias para resolver este tipo de problema. Um AG é formado de várias etapas. Algumas delas (a etapa de seleção, por exemplo) possuem diversas políticas diferentes e cada uma delas pode ser aproveitada de um problema para outro, desde que a implementação do AG esteja bem modelada.

A ferramenta aqui proposta diminui o trabalho do usuário ao desenvolver um Algoritmo Genético para um determinado problema, pois algumas das etapas do AG a ferramenta já oferece codificadas. Além disso, o modelo utilizado pela ferramenta torna a implementação do AG organizada o suficiente para evitar o "retrabalho" com a modificação de certos métodos.

Como visto neste trabalho, uma vez implementado um AG utilizando a ferramenta, é possível:

- alterar o método de seleção por uma das nove políticas de seleção disponíveis;
- alterar o método de evolução da população por uma das três políticas de evolução disponíveis;
- alterar o critério de parada por uma das cinco políticas disponíveis;
- incorporar novas políticas (seleção, evolução da população e critério de parada) à ferramenta, desde que sejam respeitados os cabeçalhos definidos para cada etapa de um AG.

A ferramenta TOGAI foi utilizada para auxiliar na implementação de AGs aplicados a dois problemas distintos encontrados na literatura: o problema da mochila e o problema da filogenia.

Para o caso de uso do problema da mochila, todo o processo de implementação do Algoritmo Genético foi realizado utilizando a ferramenta e o modelo proposto. Desta forma,

pôde-se notar que a existência de um modelo de AG pré-estabelecido facilita o trabalho de implementação, principalmente para usuários sem experiência na codificação destes algoritmos. Foi percebido ainda que a facilidade de modificação dos AGs no que diz respeito as políticas de seleção, evolução da população e critérios de parada, motivam a busca pelas melhores políticas para o problema abordado. Neste caso de uso, todas as políticas de seleção disponíveis na ferramenta foram utilizadas para encontrar o melhor resultado para o problema. Isto foi feito com facilidade, ficando o trabalho do usuário restrito às execuções dos testes com cada política.

No caso do problema da filogenia, a codificação foi derivada daquela criada por Vianna (VIANNA, 2004a) em sua tese de doutorado. A implementação do AG, para o problema, foi adaptada para que respeitasse as regras e cabeçalhos de função propostos pelo modelo. O tempo aproximado, para a adaptação do problema às regras de implementação da ferramenta, foi de uma semana. O trabalho gerado pela adaptação foi compensado pela flexibilidade adquirida no que diz respeito a futuras possibilidades de testes com diferentes políticas. Foi percebido nos testes que a política de seleção Roleta, originalmente escolhida por Vianna (VIANNA, 2004a) para compor o seu AG, não foi a que obteve o melhor desempenho. Desta forma, fazendo uso da ferramenta e das políticas melhor adaptadas ao problema, são notórias as possibilidades de melhorias de resultados. Apesar do objetivo deste caso de uso não ter sido a comparação com resultados da literatura, muitos destes foram equiparados, ajudando assim na validação da ferramenta.

A ferramenta proposta é de fácil utilização, pois gera o código na linguagem C ANSI, sem exigir que o usuário conheça técnicas de programação mais avançadas, como orientação a objetos. É extensível, porque é possível incorporar novas políticas ao modelo, tornando-o mais completo. É multiplataforma, pois, por ser desenvolvido em JAVA e gerar código em C ANSI, pode ser utilizada em diferentes tipos de máquinas sob diferentes sistemas operacionais. É eficiente, porque, embora organize o código tornando-o reutilizável, não cria restrições ao desenvolvimento do AG, podendo ser utilizado, por exemplo, qualquer operador genético sobre qualquer estrutura cromossômica. E finalmente é útil, pois o uso de AGs é uma das principais estratégias para resolver problemas difíceis de otimização combinatória.

Referências Bibliográficas

AIEX, R.; RESENDE, M. G. C.; RIBEIRO, C. C. Probability distribution of solution time in grasp: An experimental investigation. *Journal of Heuristics*, v. 8, p. 343–373, 2003.

ANDREATTA, A. A. *Uma arquitetura abstrata de domí nio para o desenvolvimento de heurí sticas de busca local com uma aplicação ao problema da filogenia*. Tese (Doutorado) — Pontifí cia Universidade Católica, Rio de Janeiro, 1998.

ANDREATTA, A. A.; RIBEIRO, C. C. Heuristics for the phylogeny problem. *Journal of Heuristics*, v. 8, p. 429–447, 2002.

ANTONISSE, J. A new interpretation of schema notation that overturns the binary encoding constraint. *Proceedings of the third international conference on Genetic algorithms*, p. 86–91, 1989.

ARAÚJO, G.; ALMEIDA, N. Phylogeny from whole genome comparison. In: *First Brasilian Workshop on Bioinformatics*. Gramado: [s.n.], 2002. p. 9 – 15.

AYALA, F. J. The myth of Eve: Molecular biology and human origins. *Science*, v. 270, p. 1930–1939, 1995.

BÄCK, T. “self adaptation in genetic algorithms” in “towards a practice on autonomous systems”. In: ____. Varela & bourgine. [S.l.]: MIT Press 1992, 1991. p. 263–271.

BAKER, J. E. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms*, p. 14–21, 1987.

BEAN, J. C. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, v. 6, p. 154–160, 1994.

BLICKLE, T.; THIELE, L. A comparison of selection schemes used in genetic algorithms. *TIK-Report nr 11 version 2. Swiss Federal Institute of Technology (ETH), Zurich*, dez. 1995.

BRINDLE, A. *Genetic algorithms for function optimization*. Dissertação (Mestrado) — University of Alberta, Edmonton, 1981.

CAHON, S.; MELAB, N.; TALBI, E. G. Paradiseo: a framework for the flexible design of parallel and distributed hybrid metaheuristics. *Journal of Heuristics*, v. 10, n. 3, p. 357–380, may 2004. ISSN 1381-1231.

CARBONO, A. J. J.; MENEZES, I. F. M. Mooring pattern optimization using genetic algorithms. *Proc. of the WCSMO6 – 6th World Congress on Structural and Multidisciplinary Optimization*, p. 182.1–182.9, 2005.

CEDENO, W.; VEMURI, V.; SLEZAK, T. Multi-niche crowding in genetic algorithms and its application to the assembly of dna restriction-fragments. *Evolutionary Computation*, v. 2, n. 4, p. 321–345, 1995.

CHOU, H.; PREMKUMAR, G.; CHU, C. H. Genetic algorithms and network design: An analysis of factors influencing ga's performance. *From the e-Business Research Center Working Paper*, p. 1–38, 1999.

CORPORATION, P. *The Guide to Evolver: The Genetic Algorithm Solver for Microsoft Excel*. Newfield, NY, 2001.

DEB, K. Multi-objective optimization using evolutionary algorithms. *John Wiley & Sons, New York*, 2001.

EDWARDS, A.; CAVALLI-SFORZA, L. Reconstruction of evolutionary trees. *Phenetic and Phylogenetic Classification*, London, v. 6, p. 67–76, 1964.

GALLUT, C.; BARRIEL, V.; VIGNES, R. Gene order and phylogenetic information. In: SANKOFF, D.; NADEAU, J. H. (Ed.). *Comparative genomics*. [S.l.]: Kluwer Academic Publishers, 2000. p. 123 – 132.

GLOVER, F. Tabu search for nonlinear and parametric optimization with links to genetic algorithms. *Discrete Applied Mathematics*, v. 49, p. 231–255, 1994.

GOLDBARG, M. C.; LUNA, H. P. L. *Otimização Combinatória e Programação Linear – Modelos e Algoritmos*. 2. ed. [S.l.]: Campus, 2000.

GOLDBERG, D. E. Genetic algorithms in search, optimization and machine learning. *Reading MA: Addison Wesley*, 1989.

GOLDBERG, D. E.; DEB, K. An investigation of niche and species formation in genetic function optimization. *Em: Proc. of International Conference on Genetic Algorithms*, p. 42–50, 1989.

GOLOBOFF, P. *Comunicação pessoal*. 1997.

GOODMAN, E. *An introduction to GALOPPS - The genetic algorithm optimized for portability and parallelism system.* [S.l.], Novembro 1994.

GREFENSTETTE, J. J. "optimization of control parameters for genetic algorithms". *IEEE - Transactions on Systems, Man & Cybernetics*, p. 122–128, 1986.

GREFENSTETTE, J. J. *A User's Guide to GENESIS Version 5.0*. Washington D.C., USA, 1990.

HENNIG, W. *Phylogenetic systematics*. Urbana: University of Illinois Press, 1966.

HERRERA, F.; LOZANO, M.; VERDEGAY, J. L. Tackling real-coded genetic algorithms: Operators and tools for the behaviour analysis. *Artificial Intelligence Review*, v. 12, p. 265–319, 1998.

HIRSCH, J. E. An index to quantify an individual's scientific research output. *Proc. Natl. Acad. Sciences (USA) 102(46) 16569-16572*, september 2005.

HOLLAND, J. H. Adaptation in natural and artificial systems. *Univ. of Michigan Press, Ann Arbor, Michigan*, 1975.

JONG, K. A. D. Analysis of the behavior of a class of genetic adaptive systems. *PhD. thesis, Dep.Computer and Comunication Sciences, Univ. Michigan, Ann Arbor*, 1975.

KITCHING, I. J. et al. *Cladistics: The theory and practice of parsimony analysis*. Londres: Oxford University Press, 1998.

LINDEN, R. *Algoritmos Genéticos*. 1. ed. [S.l.]: Brasport, 2006.

LOZANO, M.; HERRERA, F.; CANO, J. R. Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences, in press*, 2008.

LUCKOW, M.; PIMENTEL, R. A. An empirical comparison of numerical Wagner computer programs. *Cladistics*, v. 1, p. 47–66, 1985.

MARTELLO, S.; TOTH, P. *Knapsack problems: algorithms and computer implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN 0-471-92420-2.

MICHALEWICZ, Z. *Genetic Algorithms+Data Structures = Evolution Programs*. [S.l.]: Springer-Verlag, 1994.

OLIVER, I. M.; SMITH, D. J.; HOLLAND, J. R. C. "a study of permutation crossover operators on the traveling salesman problemm". *Proceedings of Second International Conference on Genetic Algorithms*, p. 224–230, 1987.

PENNY, D.; FOULDS, L. R.; HENDY, M. D. Testing the theory of evolution by comparing phylogenetic trees constructed from five different protein sequences. *Nature*, v. 247, p. 197–200, 1982.

PISINGER, D. Core problems in knapsack algorithms. *Oper. Res.*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 47, n. 4, p. 570–575, 1999. ISSN 0030-364X.

PISINGER, D. Where are the hard knapsack problems? *Comput. Oper. Res.*, Elsevier Science Ltd., Oxford, UK, UK, v. 32, n. 9, p. 2271–2284, 2005. ISSN 0305-0548.

PISINGER, D.; MARTELLO, S.; TOTH, P. *New Trends in Exact Algorithms for the 0-1 Knapsack Problem*. Denmark, 1997.

PLATNICK, N. I. An empirical comparison of microcomputer parsimony programs. *Cladistics*, v. 3, p. 121–144, 1987.

PLATNICK, N. I. An empirical comparison of microcomputer parsimony programs II. *Cladistics*, v. 5, p. 145–161, 1989.

POHLHEIM, H. Geatbx-genetic and evolutionary algorithm toolbox for matlab. 2005. Disponível em: <http://www.geatbx.com/>.

RADCLIFFE, N. J. Nonlinear genetic representations. *Parallel Problem Solving from Nature 2, R.Manner and B. Manderick*, p. 259–268, 1992.

RIBEIRO, C. C.; VIANNA, D. S. A genetic algorithm for the phylogeny problem using an optimized crossover strategy based on path-relinking. *Revista Tecnologia da Informação - RTInfo*, v. 3, p. 67–70, 2003.

SCHAFFER, J. D. et al. A study of control parameters affecting online performance of genetic algorithms for function optimization. In: *Proceedings of the third international conference on Genetic algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 51–60. ISBN 1-55860-006-3.

SCHWEFEL, H. P. *“On the evolution of evolutionary computation”, Computational Intelligence: Imitating Life*. Piscataway, NJ: by J. M. Zurada, R. J. Marks II, and C. J. Robinson IEEE Press, 1994. 116–124 p.

SOARES, G. L. *Algoritmo Genético: Estudo, Novas Técnicas e Aplicações*. Dissertação (Mestrado) — UFMG, Belo Horizonte, 1997.

SOBER, E. Parsimony likelihood and the principle of the common cause. *Philosophy of Science*, v. 54, p. 465–469, 1987.

STAA, A. V. Regras e recomendações para a programação em c e c++. *Versão 1.01. 18 p. Port. 2000*, 2000.

SWOFFORD, D. L.; OLSEN, G. Phylogeny reconstruction. In: HILLIS, D. M.; MORITZ, C. (Ed.). *Molecular systematics*. [S.l.]: Sinauer, 1990. p. 411–501.

SYSWERDA, G. Uniform crossover in genetic algorithms. *In J. D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann*, p. 2–9, 1989.

TALIGENT, I. Leveraging object-oriented frameworks. *A Taligent White Paper*, 1993.

TALIGENT, I. Building object-oriented frameworks. *A Taligent White Paper*, 1994.

TOSCANI, L. V.; VELOSO, P. A. S. *Complexidade de Algoritmos: análise, projeto e métodos*. 2. ed. Porto Alegre: Sagra Luzzato, 2005.

VASCONCELOS, J. A. et al. Genetic algorithm coupled with a deterministic method for optimization in electromagnetics. *IEEE Transactions on Magnetics*, v. 32, n. 2, p. 1860–1863, 1997.

VASCONCELOS, J. A.; TAKAHASHI, R. H. C.; SALDANHA, R. R. Improvements in genetic algorithms. *IEEE - Transactions on Magnetics*, v. 37, n. 5, 2001.

VIANNA, D. S. *Heurísticas híbridas para o problema da filogenia*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, fevereiro 2004.

VIANNA, D. S. *Heurísticas Híbridas para o Problema da Filogenia*. Tese (Doutorado) — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2004.

VIANNA, D. S.; OCHI, L. S.; DRUMMOND, L. M. A. A parallel hybrid evolutionary metaheuristic for the period vehicle routing problem with heterogeneous fleet. *Lecture Notes in Computer Science*, v. 1388, p. 216–225, 1999.

VOGEL, G. Phylogenetic analysis: Getting its day in court. *Science*, v. 275, p. 1559–1560, 1997.

WALL, M. *GAlib: A C++ Library of Genetic Algorithm Components*. [S.l.], Agosto 1996. Disponível em: <http://lancet.mit.edu/ga/dist/galibdoc.pdf>.

WANG, L.; WARNOW, T. New polynomial-time methods for whole-genome phylogeny reconstruction. In: *33rd Symposium on Theory of Computation*. Crete: [s.n.], 2001. p. 637 – 646.

WILEY, E. O. et al. *The Complete Cladistics: A Primer of Phylogenetic Procedures*. Museum of Natural History, 1991.

WILSON, M. V. H. Importance for phylogeny of single and multiple stem-group fossil species with examples from freshwater fishes. *Systematics Biology*, v. 41, p. 462–470, 1992.

Apêndice A – Código Fonte da Ferramenta TOGAI

A ferramenta TOGAI utiliza trechos de código na linguagem de programação C ANSI para gerar os Algoritmos Genéticos. Serão apresentados a seguir o código fonte da estrutura principal e das políticas de seleção, evolução da população e critério de parada.

A.1 Código fonte da estrutura principal

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#include<math.h>

/* Constante com o maior valor suportado pelo tipo de variável que armazena a função objetivo */
#define INFINITE_MORE (float)3.4e+38
#define INFINITE_LESS (float)-3.4e+38
/* O valor 1 da constante MAXIMIZATION deverá ser alterado para 0 caso seu problema seja de minimização */
#define MAXIMIZATION 1

/* Macro para retornar o melhor valor entre dois comparados */
#if MAXIMIZATION
    #define best(A,B) ((A>B) ? (1):(0))
#else
    #define best(A,B) ((A<B) ? (1):(0))
#endif

/* Constante que armazena o pior valor de função objetivo (para inicialização de variáveis) */
#define OBJ_VAL_INI (best(INFINITE_LESS,INFINITE_MORE) ? (INFINITE_MORE):(INFINITE_LESS))
/* Constante que armazena o melhor valor possível de função objetivo (para inicialização de variáveis) */
#define OBJ_VAL_END (best(INFINITE_LESS,INFINITE_MORE) ? (INFINITE_LESS):(INFINITE_MORE))

/******************************************************************************
* Constantes que armazenam parâmetros comumente definidos em AGs
* Poderão ser modificadas de acordo com as necessidades do usuário
******************************************************************************/
#define SIZE_POP 100    /* Número de indivíduos da população */
#define TOT_SELECT 2    /* Total de indivíduos selecionados para cruzamento */
#define TOT_CHILDREN 2  /* Total de filhos gerados em um cruzamento */
#define MAX_GEN 1000    /* Número máximo de gerações para o algoritmo */
#define PROB_MUTAT 0.1  /* Valor da probabilidade de mutação */
#define PROB_CROSS 0.8  /* Valor da probabilidade de cruzamento */
/******************************************************************************
*FIM: declaração de parâmetros dos AGs
******************************************************************************/

/* Estrutura do cromossomo */
typedef struct chromossome
{
    /* Deverá ser incluída aqui a estrutura de dados do cromossomo
    Ex:  O cromossomo poderá ser formado por 10 elementos inteiros*/
    int iAlele[10];
} tpChromossome;

/* Deverá ser definido aqui o tipo de dado que guarda o valor da função objetivo (tpObj) */
/* Tipo padrão: float. */
typedef float tpObj;

/* Estrutura dos indivíduos: formada pelos cromossomos e pelos valores de suas funções objetivo. */
```

```
typedef struct individual
{
    tpChromossome chrom;
    tpObj obj;
} tpIndivid;

/* Definição do tipo da população como um vetor de individuos */
typedef tpIndivid tpPopulation[SIZE_POP];

/******************************************************************************
* Protótipos das funções dependentes da estrutura cromossômica
* Obs:Os conteúdos destas funções deverão ser implementados pelo usuário.
*******************************************************************************/
void copyChrom( tpChromossome*, tpChromossome* );
void allocIndivid( tpIndivid* );
void createIndivid( tpIndivid* );
void deallocIndivid( tpIndivid* );
void crossover( tpPopulation, int*, tpIndivid* );
void mutation( tpChromossome* );
void verifyChromChild( tpChromossome* );
void calcObj ( tpIndivid* );
/******************************************************************************
* Protótipos das funções independentes da estrutura cromossômica
* Obs:O conteúdo destas funções não necessitam ser alterados pelo usuário.
*******************************************************************************/
void initialPopulation( tpPopulation );
void deallocPopulation( tpPopulation );
void copyIndivid( tpIndivid *pDestin, tpIndivid *pOrigin );
void selection( tpPopulation, int* );
tpPopulation* populationEvolution ( tpPopulation, tpIndivid* );
void updatePop( tpPopulation, tpIndivid*, int* );
tpPopulation* stopCriterion ( tpPopulation, tpIndivid* );
void keepFather( tpPopulation, int*, tpIndivid* );
void bestSolut( tpIndivid*, tpIndivid* );
/*******************************************************************************
*FIM: declaração dos protótipos das funções
*******************************************************************************/

/******************************************************************************
* Título: COPIA CROMOSSOMO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Implementa a cópia do conteúdo do cromossomo de um indivíduo para
*            um outro.
*
* Entrada:   *pDestin - Ponteiro com o endereço da estrutura do cromossomo
*                       que será substituído.
*            *pOrigin - Ponteiro com o endereço da estrutura do cromossomo
*                       que será copiado.
* Saída:     É copiado o cromossomo do endereço de origem para o de destino
*******************************************************************************/
void copyChrom( tpChromossome *pDestin, tpChromossome *pOrigin )
{

}
/*******************************************************************************
*FIM: COPIA CROMOSSOMO
*******************************************************************************/

/******************************************************************************
* Título: ALOCA INDIVÍDUO DA MEMÓRIA (Obs: pode não ser necessária alteração)
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Aloca um indivíduo da população.
*
* Entrada:   *pIndivid - Ponteiro com o endereço do indivíduo que será desalo-
*                        cado da memória
* Saída:     O indivíduo é desalocado, liberando a memória
*******************************************************************************/
void allocIndivid( tpIndivid *pIndivid )
{

}
/*******************************************************************************
*FIM: ALOCA INDIVÍDUO DA MEMÓRIA
*******************************************************************************/

/******************************************************************************
* Título: CRIAÇÃO DE INDIVÍDUO DA POPULAÇÃO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Implementa a criação de um indivíduo para a população inicial.
*
```

```
* Entrada:   *pIndivid - Ponteiro com o endereço onde será alocado o novo
*                        indivíduo da população
* Saída:     Um indivíduo gerado para a população inicial
*************************************************************************/
void createIndivid( tpIndivid *pIndivid )
{

}
/*************************************************************************
*FIM: CRIAÇÃO DE INDIVÍDUO DA POPULAÇÃO
*************************************************************************/

/*************************************************************************
* Título: DESALOCA INDIVÍDUO DA MEMÓRIA (Obs: pode não ser necessária alteração)
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Desaloca um indivíduo da população.
*
* Entrada:   *pIndivid - Ponteiro com o endereço do indivíduo que será desalo-
*                        cado da memória
* Saída:     O indivíduo é desalocado, liberando a memória
*************************************************************************/
void deallocIndivid( tpIndivid *pIndivid )
{

}
/*************************************************************************
*FIM: DESALOCA INDIVÍDUO DA MEMÓRIA
*************************************************************************/

/*************************************************************************
* Título: CRUZAMENTO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Implementa o cruzamento entre indivíduos da população.
*
* Entrada:   pop -         Estrutura que armazena os cromossomos e funções
*                          objetivos dos indivíduos da população
*            *pSelected - Vetor que possui a identificação dos indivíduos que
*                          foram selecionados para o cruzamento
*            *pChildren - Vetor que armazena os filhos gerados no cruzamento
* Saída:     O vetor *pChildren com os filhos gerados no cruzamento
*************************************************************************/
void crossover( tpPopulation pop, int *pSelected, tpIndivid *pChildren )
{

}
/*************************************************************************
*FIM: CRUZAMENTO
*************************************************************************/

/*************************************************************************
* Título: MUTAÇÃO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Implementa uma alteração na composição do cromossomo de um
*            indivíduo da população, visando a diversidade genética.
*
* Entrada:   *pChromChild - Ponteiro que aponta para a estrutura do cromossomo
*                           de um descendente, gerado no último cruzamento,
*                           que sofrerá a mutação
* Saída:     A estrutura do cromossomo com o seu conteúdo alterado
*************************************************************************/
void mutation(tpChromossome *pChromChild)
{

}
/*************************************************************************
*FIM: MUTAÇÃO
*************************************************************************/

/*************************************************************************
* Título: VALIDAÇÃO DE SOLUÇÃO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Verifica a validade de uma solução e caso a mesma seja inviável
*            utiliza um algoritmo para fazer a validação.
*
* Entrada:   *pChromChild - Ponteiro que aponta para a estrutura do cromossomo
*                           de um descendente, gerado no último cruzamento,
*                           que será validado
* Saída:     A estrutura do cromossomo com o seu conteúdo validado
```

```
******************************************************************************/
void verifyChromChild(tpChromossome *pChromChild)
{

}
/******************************************************************************
*FIM: VALIDAÇÃO DE SOLUÇÃO
******************************************************************************/

/******************************************************************************
* Título: CÁLCULO DE FUNÇÃO OBJETIVO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Calcula o valor da função objetivo de um indivíduo.
*
* Entrada:   *pChildren - Ponteiro que aponta para a estrutura do indivíduo
*                         que terá sua função objetivo calculada
* Saída:     O campo que armazena a função objetivo do indivíduo preenchido
******************************************************************************/
void calcObj( tpIndivid *pChildren )
{

}
/******************************************************************************
*FIM: CÁLCULO DE FUNÇÃO OBJETIVO
******************************************************************************/

/******************************************************************************/
/******************************************************************************/
/******************************************************************************/
/******************************************************************************/
/* Área onde se encontram as funções independentes da estrutura do cromossomo */
/*          (não necessitam de alteração por parte do usuário)               */
/******************************************************************************/
/******************************************************************************/
/******************************************************************************/
/******************************************************************************/


/******************************************************************************
* Título: POPULAÇÃO INICIAL
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Geração da população inicial. Cada indivíduo da população tem a
*            sua estrutura de dados enviada para a função createIndivid que
*            gera um novo indivíduo na população.
*
* Entrada:   pop         - Estrutura que armazena os cromossomos e funções
*                          objetivos dos indivíduos da população
* Saída:     É gerada a população inicial em na estrutura pop
******************************************************************************/
void initialPopulation( tpPopulation pop )
{
    int idIndiv;  /* Identificador do indivíduo da população*/

    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        createIndivid( &pop[idIndiv] );
    }
}
/******************************************************************************
*FIM: POPULAÇÃO INICIAL
******************************************************************************/

/******************************************************************************
* Título: DESALOCA POPULAÇÃO
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Desaloca da memória a estrutura de dados da população
*
* Entrada:   pop         - Estrutura que armazena os cromossomos e funções
*                          objetivos dos indivíduos da população
* Saída:     É gerada a população inicial em na estrutura pop
******************************************************************************/
void deallocPopulation( tpPopulation pop )
{
    int idIndiv;  /* Identificador do indivíduo da população*/

    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        deallocIndivid( &pop[idIndiv] );
    }
    free(pop);
```

```
}
/***************************************************************************
*FIM: DESALOCA POPULAÇÃO
***************************************************************************/

/***************************************************************************
* Título: COPIA INDIVIDUO
* Versão: ?
* Autor: ?
* Data: ?
* Descrição: Implementa a cópia de um indivíduo para outro.
*
* Entrada:   *pDestin - Ponteiro com o endereço da estrutura do indivíduo
*                       que será substituído.
*            *pOrigin - Ponteiro com o endereço da estrutura do indivíduo
*                       que será copiado.
* Saída:     É copiado o indivíduo do endereço de origem para o de destino
***************************************************************************/
void copyIndivid( tpIndivid *pDestin, tpIndivid *pOrigin )
{
copyChrom(&(pDestin->chrom), &(pOrigin->chrom));
pDestin->obj = pOrigin->obj;
}

/***************************************************************************
*FIM: Copia Individuo
***************************************************************************/

/***************************************************************************
* Título: MANTER PAIS
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que é utilizada para, caso a probabilidade de cruzamento
*            não seja atendida, manter na nova população os melhores pais
*            daqueles selecionados para cruzamento.
*
* Entrada: pop        - Estrutura que armazena os cromossomos e funções
*                       objetivos dos indivíduos da população
*          *pSelected - Vetor que será retornado com a identificação dos
*                       indivíduos selecionados
*          *pChildren - Ponteiro que aponta para a estrutura onde estão os
*                       filhos gerados por um cruzamento
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
***************************************************************************/
void keepFather( tpPopulation pop, int *pSelected, tpIndivid *pChildren )
{
    int   idChild,  /* Identificação do filho na estrutura de filhos gerada pelo cruzamento */
          idFather, /* Identificação do pai na estrutura de pais gerada pela seleção */
          idBetter; /* Identificação do melhor indivíduo em um conjunto */
    tpObj betterObj,/* Melhor valor de função objetivo dentre um grupo de indivíduos */
          prevObj;  /* Melhor valor de função objetivo computada anteriormente para um grupo de indivíduos */

    prevObj = OBJ_VAL_END;
    /* Loop controlado pelo número de filhos que serão, na verdade, preenchidos pelos melhores pais */
    for ( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
    {
        betterObj = OBJ_VAL_INI;
        /* Loop controlado pelo número de indivíduos anteriormente selecionados */
        for ( idFather = 0; idFather < TOT_SELECT; idFather++ )
        {
            /* Comparações para encontrar o melhor pai dos ainda não selecionados (piores do que prevObj)*/
            if  (  best( pop[pSelected[idFather]].obj, betterObj ) )
            {
                if ( best( prevObj, pop[pSelected[idFather]].obj ) )
                {
                    betterObj = pop[pSelected[idFather]].obj;
                    idBetter = pSelected[idFather];
                }
            }
        }
        prevObj = betterObj;

        pChildren[idChild].obj = betterObj;
        /* Loop para copiar o cromossomo da estrutura do pai para o filho */

        copyChrom( &pChildren[idChild].chrom, &pop[idBetter].chrom );

        /* Se o número de filhos for maior do que o de pais o processo reinicia copiando o melhor pai */
        if ( idChild == idFather - 1 )
        {
            prevObj = OBJ_VAL_END;
        }
    }
```

```
}
/*******************************************************************************
*FIM: MANTER PAIS
*******************************************************************************/

/*******************************************************************************
* Título: MELHOR SOLUÇÃO
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que compara um indivíduo com o melhor indivíduo gerado até
*            aquele momento e, caso o atual seja melhor, substitui o indivíduo
*            existente em *bestIndivid pelo atual.
*
* Entrada:*pChildren   - Ponteiro que aponta para a estrutura onde estão os
*                        filhos gerados por um cruzamento
*         *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                        indivíduo gerado pelo algoritmo genético
*
* Saída:   A melhor solução para o problema (*bestIndivid) até aquela geração
*******************************************************************************/
void bestSolut( tpIndivid *pChildren, tpIndivid *bestIndivid )
{
    if(best((*pChildren).obj, (*bestIndivid).obj))
{
(*bestIndivid).obj=(*pChildren).obj; /* Copiar o valor da função objetivo da estrutura do filho para a estrutura better c
copyChrom(&bestIndivid[0].chrom, &pChildren[0].chrom);/* Copiar o cromossomo da estrutura do filho para a estrutura bette
}
}
/*******************************************************************************
*FIM: MELHOR SOLUÇÃO
*******************************************************************************/

/*******************************************************************************
* Título: PROGRAMA PRINCIPAL
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Programa integrante da ferramenta TOGAI, que visa facilitar a
*            implementação de algoritmos genéticos mono objetivos na linguagem C
*            ANSI
*
* Entrada: Instancia com os dados do problema (via arquivo texto)
* Saída:   A melhor solução para o problema do usuário
*******************************************************************************/
main( )
{
    tpIndivid      bestIndivid;/* Melhor indivíduo */
    tpPopulation   *pop;        /* Estrutura que armazena os indivíduos da população */

    pop = ( tpPopulation* ) malloc ( sizeof ( tpPopulation ) );
    bestIndivid.obj = OBJ_VAL_INI; /* Atribui o pior valor possível a variável */
    srand( time( NULL ) );         /* Gera uma semente aleatória */
    initialPopulation( pop );      /* Função que gera a população inicial */
    //pop = stopCriterion ( pop, &bestIndivid ); /* Inicia a execução do AG */
    /* Resultado da execução do AG - o tipo da variável deverá ser revisto */
    printf("Melhor valor de função objetivo: %.4f", bestIndivid.obj);
    deallocPopulation( pop );
}
/*******************************************************************************
*FIM: PROGRAMA PRINCIPAL
*******************************************************************************/

/*|END SECTION|*/
```

A.2 Código fonte das políticas de seleção

```
/**************************************************************************
* Título: SELEÇÃO - DETERMINÍSTICA
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Uma população auxiliar será gerada com o número de cópias que um
*            indivíduo tiver direito. Esse número será calculado através da
*            divisão do valor da função objetivo do indivíduo pela média das
*            funções objetivo dos demais indivíduos (fi/fmed).
*            As vagas não preenchidas pelo cálculo anterior, serão preenchidas
*            pelos indivíduos que tiverem como resultado da divisão as melhores
*            partes fracionárias.
*            Uma vez completada a população, uma seleção aleatória será feita
*            para encontrar um novo pai que fará parte do próximo cruzamento.
*
* Entrada: pop       - Estrutura que armazena os cromossomos e funções
*                      objetivos dos indivíduos da população
*          pSelected - Vetor que será retornado com a identificação dos
*                      indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
**************************************************************************/
void selection(tpPopulation pop, int *pSelected)
{
    int   idIndiv,                     /* Identificador do indivíduo da população*/
          idNewFather,                 /* Indice que referencia os indivíduos selecionados (novos pais)*/
          iIndivFracBttr,              /* Indivíduo de melhor parte fracionária */
          iPosition        = 0,        /* Posição que será ocupada pelo indivíduo na população auxiliar */
          iTotCopies       = 0,        /* Total de cópias que um indivíduo tem direito na população auxiliar */
          iCopy;                       /* Variável para controle de iterações de loops */
    int   vtPopAux[SIZE_POP];          /* Vetor que armazena a população auxiliar */
    float fFracBetter,                 /* Melhor parte fracionária entre os indivíduos */
          fFracCurrent,                /* Valor fracionário do indivíduo atual */
          fFracPrevious    = 0.999999,/* Último valor de parte fracionária já escolhido */
          fAddObj          = 0,        /* Soma de todos os valores das funções objetivo */
          fAverageObj;                 /* Média dos valores das funções objetivo dos indivíduos da população */
    tpObj greatValue;                  /* Utilizado para calcular o número de cópias do indivíduo caso o problema seja d


    greatValue = pop[0].obj;
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        fAddObj = fAddObj + pop[idIndiv].obj;
        if ( pop[idIndiv].obj > greatValue ) /* Usado quando o problema é de minimização */
        {
            greatValue = pop[idIndiv].obj;
        }
    }

    /* Cálculo da média dos valores das funções objetivo dos indivíduos */
    fAverageObj = fAddObj / SIZE_POP;
    /* Inclusão das cópias dos indivíduos na população auxiliar */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( MAXIMIZATION )
        {
            iTotCopies = ( int )( pop[idIndiv].obj / fAverageObj );
        }
        else
        {
            iTotCopies = ( int )( ( greatValue - pop[idIndiv].obj ) / fAverageObj );
        }
        if ( iTotCopies >= 1 )
        {
            for ( iCopy = 1; iCopy <= iTotCopies; iCopy++ )
            {
                vtPopAux[iPosition] = idIndiv;
                iPosition++;
            }
        }
    }
    /* Preenchimento das posições ainda vazias da população auxiliar com os
       indivíduos de melhor parte fracionária.*/
    fFracPrevious = 0.999999;
    for ( iCopy = iPosition; iCopy < SIZE_POP; iCopy++ )
    {
        fFracCurrent = ( pop[0].obj/fAverageObj ) - ( int )( pop[0].obj/fAverageObj );
        fFracBetter = fFracCurrent;
        iIndivFracBttr = 0;
        for ( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
        {
```

```
            fFracCurrent = ( pop[idIndiv].obj/fAverageObj ) - ( int )( pop[idIndiv].obj/fAverageObj );
            if  ( best( fFracCurrent , fFracBetter ) && best( fFracPrevious , fFracCurrent ) )
            {
                fFracBetter = fFracCurrent;
                iIndivFracBttr = idIndiv;
            }
        }
        /* Indivíduo de melhor valor na parte fracionária é atribuído a
           próxima posição vazia da população auxiliar */
        vtPopAux[iPosition] = iIndivFracBttr;
        iPosition++;
        fFracPrevious = fFracBetter;
    }
    /* Seleção aleatória de um indivíduo da população temporária */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = vtPopAux[rand(  ) % SIZE_POP];
    }
}
/******************************************************************************
*Fim: DETERMINÍSTICA
******************************************************************************/
```

```
/*****************************************************************************
* Título: SELEÇÃO - TORNEIO - Variação: ROLETA
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Nesta função um subconjunto de indivíduos é escolhido utilizando
*            o método da roleta, contudo somente o individuo com maior valor
*            para a função de adequação será selecionado.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através de * pSelected
*****************************************************************************/
#define NORM 10000 /* Normalizador para geração do fitness - Roleta */
void roletaTorn( tpPopulation, int , int* );
#define TOT_COMPETIT 3 /* Total de competidores para participar do torneio */
void selection( tpPopulation pop, int* pSelected )
{
    int   idIndiv,           /* Identificador do indivíduo da população*/
          idNewFather,       /* Indice que referencia os indivíduos selecionados (novos pais)*/
          iBetterIndiv,      /* Melhor indivíduo do subconjunto sorteado */
          iCompetitor;       /* Indice que referencia os competidores sorteados*/
    int*  pSelectedRoulette; /* Vetor que armazenará os indivíduos selecionados pela roleta*/
    tpObj betterObj;         /* Armazena o melhor valor de função objetivo */

    /* Estrutura de dados que irá receber os competidores selecionados pela roleta */
    pSelectedRoulette = malloc( TOT_COMPETIT * sizeof( int ) );

    /* Repetição a ser feita um número de vezes igual a quantidade de indivíduos
       a serem selecionados pela função */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Chamada a função roleta para seleção dos competidores do torneio*/
        roletaTorn( pop, TOT_COMPETIT, pSelectedRoulette );

        /* Escolha do melhor indivíduo dada sua função objetivo */
        idIndiv = pSelectedRoulette[0];
        betterObj = pop[idIndiv].obj;
        iBetterIndiv = idIndiv;
        for ( iCompetitor = 1; iCompetitor < TOT_COMPETIT; iCompetitor++ )
        {
            idIndiv = pSelectedRoulette[iCompetitor];
            if ( best( pop[idIndiv].obj , betterObj )
            {
                betterObj = pop[idIndiv].obj;
                iBetterIndiv = idIndiv;
            }
        }
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = iBetterIndiv;
    }
    free( pSelectedRoulette );
}
/*****************************************************************************
* Título: ROLETA - (Parte da política de seleção TORNEIO - Variação: ROLETA)
* Versao: 1.0
*  Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Implementação da política de seleção ROLETA na sua forma
*            tradicional.Os valores das funções objetivo dos indivíduos são
*            utilizados como setores de uma roleta, em seguida a roleta é
*            girada para que os indivíduos sejam selecionados e retornados para
*            a função de seleção Torneio.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*          iTotCompetit- Total de indivíduos a serem selecionados para competir
*                        no torneio
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através de * pSelected
*****************************************************************************/
void roletaTorn( tpPopulation pop, int iTotCompetit, int* pSelected )
{
    int      idIndiv,                /* Identificador do indivíduo da população*/
             idNewFather;            /* Indice que referencia os indivíduos selecionados (novos pais)*/
    long int lRandom;                /* Armazena o número randomico gerado */
    float    vtFitness[SIZE_POP],    /* Vetor para armazenamento do fitness */
             fAddFitness          = 0,/* Armazena o somatório do fitness dos indivíduos */
```

```
            iTurn                = 0;/* Controla o giro da roleta */
    tpObj    betterObj,              /* Armazena o melhor valor de função objetivo */
             greatValue;             /* Utilizado na criação da roleta caso o problema seja de minimização */
    /* Identificação do maior valor entre as funções objetivo para cálculo do
       fitness */
    betterObj = pop[0].obj;
    greatValue = pop[0].obj;
    for ( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( pop[idIndiv].obj > betterObj )
        {
            betterObj = pop[idIndiv].obj;
        }
    }
    /* Cálculo e somatório do fitness */
    if ( MAXIMIZATION )
    {
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = (pop[idIndiv].obj * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    else
    {
        for( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( pop[idIndiv].obj > greatValue )
            {
                greatValue = pop[idIndiv].obj;
            }
        }
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = ( ( greatValue - pop[idIndiv].obj ) * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    /* Seleção dos indivíduos de acordo com o número de pais a serem
       selecionados */
    for ( idNewFather = 0; idNewFather < iTotCompetit; idNewFather++ )
    {
        /* Giro da roleta */
        idIndiv = 0;
        iTurn = vtFitness[idIndiv];
        lRandom = ( ( rand(  ) * rand(  ) ) % ( long int )fAddFitness );
        while ( iTurn < lRandom )
        {
            idIndiv++;
            iTurn = iTurn + vtFitness[idIndiv];
        }
        /* Atribuição de um novo indivíuo selecionado como pai à estrutura que
           será retornada */
        pSelected[idNewFather] = idIndiv;
    }
}
/******************************************************************************
*Fim: TORNEIO - Variação: ROLETA
******************************************************************************/
```

```
/****************************************************************************
* Título: SELEÇÃO - RANKING
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Inicialmente os indivíduos da população são ordenados de acordo
*            com os valores das suas funções de adaptação. Após a ordenação,
*            cada indivíduo terá um valor equivalente a sua posição no ranking
*            e, um procedimento similar à seleção pelo método da roleta será
*            utilizado. Quanto melhor a posição do indivíduo no ranking, maior
*            será seu setor na roleta e conseqüentemente maior a sua chance de
*            ser selecionado.
*
* Entrada:   pop        - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*            pSelected  - Vetor que será retornado com a identificação dos
*                         indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
****************************************************************************/
#define NORM 10000 /* Normalizador para geração do fitness - Roleta */
void roletaRank(int* , int* );
void selection( tpPopulation pop, int* pSelected )
{
    int   idIndiv,          /* Identificador do indivíduo da população */
          idIndiv2,         /* Identificador de indivíduo auxiliar para comparação */
          vtRank[SIZE_POP]; /* Vetor que armazena o rank gerado para cada indivíduo */

    /* Inicialização do vetor de rank */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        vtRank[idIndiv] = 1;
    }
    /* Cadastro dos ranks no vetor */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        for ( idIndiv2 = 0; idIndiv2 < SIZE_POP; idIndiv2++ )
        {
            if ( best( pop[idIndiv].obj , pop[idIndiv2].obj ) )
            {
                vtRank[idIndiv]++;
            }
        }
    }
    roletaRank( vtRank, pSelected );
}

/****************************************************************************
* Título: ROLETA - (Parte da política de seleção por RANKING)
* Versao: 1.0
*  Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Implementação da política de seleção ROLETA na sua forma
*            tradicional.Os valores gerados pela função RANKING são
*            utilizados como setores de uma roleta, em seguida a roleta é
*            girada para que os indivíduos sejam selecionados e retornados para
*            cruzamento.
*
* Entrada: * vetRank   - Estrutura que armazena o rank gerado para cada um dos
*                        indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através de * pSelected
****************************************************************************/
void roletaRank(  int* vetRank,  int* pSelected )
{
    int      idIndiv,        /* Identificador do indivíduo da população */
             idNewFather;    /* Indice que referencia os indivíduos selecionados (novos pais)*/
    long int lAddRank    = 0,/* Somatório dos ranks */
             lRandom;        /* Armazena o número randomico gerado */
    float    iTurn       = 0;/* Controla o giro da roleta */

    /* Somatório dos ranks */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        lAddRank = lAddRank + vetRank[idIndiv];
    }
    /* Seleção dos indivíduos de acordo com o número de pais a serem
       selecionados */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        idIndiv = 0;
        iTurn = vetRank[idIndiv];
```

```
        lRandom = ( ( rand(  ) * rand(  ) ) % lAddRank );
        /* Giro da roleta */
        while ( iTurn < lRandom )
        {
            idIndiv++;
            iTurn = iTurn + vetRank[idIndiv];
        }
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = idIndiv;
    }
}
/********************************************************************************
*Fim: RANKING
********************************************************************************/
```

```
/*****************************************************************************
* Título: SELEÇÃO - ROLETA
* Versao: 1.0
*  Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Implementação da política de seleção ROLETA na sua forma
*            tradicional.Os valores das funções objetivo dos indivíduos são
*            utilizados como setores de uma roleta, em seguida a roleta é
*            girada para que os indivíduos sejam selecionados.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através de * pSelected
*****************************************************************************/
#define NORM 10000 /* Normalizador para geração do fitness - Roleta */
void selection( tpPopulation pop, int *pSelected )
{
int       idIndiv,                  /* Identificador do indivíduo da população*/
              idNewFather;              /* Indice que referencia os indivíduos selecionados (novos pais)*/
long int  lRandom;                  /* Armazena o número randômico gerado */
float     vtFitness[SIZE_POP],      /* vetor para armazenamento do fitness */
  fAddFitness           = 0, /* Armazena o somatório do fitness dos indivíduos */
  iTurn                 = 0; /* Controla o giro da roleta */
tpObj     betterObj,                /* Armazena o melhor valor de função objetivo */
              greatValue;               /* Utilizado na criação da roleta caso o problema seja de minimização */

/* Identificação do maior valor entre as funções objetivo para cálculo
   do fitness */
betterObj = pop[0].obj;
greatValue = pop[0].obj;
for( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
{
        if( best( pop[idIndiv].obj , betterObj ))
        {
       betterObj = pop[idIndiv].obj;
        }
    }
    /* Cálculo e somatório do fitness */
    if ( MAXIMIZATION )
    {
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = (pop[idIndiv].obj * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    else
    {
        for( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( pop[idIndiv].obj > greatValue )
            {
                greatValue = pop[idIndiv].obj;
            }
        }
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = ( ( greatValue - pop[idIndiv].obj ) * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    /* Seleção dos indivíduos de acordo com o número de pais a serem
   selecionados */
    for( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++)
{
        /* Giro da roleta */
idIndiv = 0;
iTurn = vtFitness[idIndiv];
lRandom = ( rand(  ) * rand(  ) ) % (long int)fAddFitness );
while( iTurn < lRandom )
{
            idIndiv++;
            iTurn = iTurn + vtFitness[idIndiv];
        }
        /* Atribuição da identificação do indivíduo selecinado a estrutura
           que será acessada pela função chamadora*/
        pSelected[idNewFather] = idIndiv;
    }
}
/*****************************************************************************
*FIM: ROLETA
*****************************************************************************/
```

```
/****************************************************************************
* Título: SELEÇÃO - ESTOCÁSTICA POR RESTO SEM REPOSIÇÃO
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Uma população auxiliar será gerada com o número de cópias que um
*            indivíduo tiver direito. Esse número será calculado através da
*            divisão do valor da função objetivo do indivíduo pela média das
*            funções objetivo dos demais indivíduos (fi/fmed).
*            As vagas não preenchidas pelo cálculo anterior, serão preenchidas
*            mediante uma escolha aleatória de um indivíduo, que terá sua parte
*            fracionária usada como porcentagem de possibilidade para poder
*            compor a população auxiliar.
*            Uma vez comletada a população, uma seleção aleatória será feita
*            para encontrar um novo pai que fará parte do próximo cruzamento.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
****************************************************************************/
void selection(tpPopulation pop, int* pSelected)
{
    int   idIndiv,                 /* Identificador do indivíduo da população*/
          idNewFather,             /* Indice que referencia os indivíduos selecionados (novos pais)*/
          iPosition          = 0,/* Posição que será ocupada pelo indivíduo na população auxiliar */
          iTotCopies         = 0,/* Total de cópias que um indivíduo tem direito na população auxiliar */
          iCopy,                   /* Variável para controle de iterações de loops */
          iDraw,                   /* Sorteio aleatório para comparação com a probabilidade de seleção do indivíduo*/
          vtPopAux[SIZE_POP];      /* Vetor que armazena a população auxiliar */
    float fFracCurrent,            /* Valor fracionário do indivíduo atual */
          fAddObj            = 0,/* Soma de todos os valores das funções objetivo */
          fAverageObj;             /* Média dos valores das funções objetivo dos indivíduos da população */
    tpObj greatValue;              /* Utilizado para calcular o número de cópias do indivíduo caso o problema seja de mini

    greatValue = pop[0].obj;
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        fAddObj = fAddObj + pop[idIndiv].obj;
        if ( pop[idIndiv].obj > greatValue ) /* Usado quando o problema é de minimização */
        {
            greatValue = pop[idIndiv].obj;
        }
    }

    /* Cálculo da média dos valores das funções objetivo dos indivíduos */
    fAverageObj = fAddObj / SIZE_POP;
    /* Inclusão das cópias dos indivíduos na população auxiliar */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( MAXIMIZATION )
        {
            iTotCopies = ( int )( pop[idIndiv].obj / fAverageObj );
        }
        else
        {
            iTotCopies = ( int )( ( greatValue - pop[idIndiv].obj ) / fAverageObj );
        }
        if ( iTotCopies >= 1 )
        {
            for ( iCopy = 1; iCopy <= iTotCopies; iCopy++ )
            {
                vtPopAux[iPosition] = idIndiv;
                iPosition++;
            }
        }
    }
    /* Preenchimento das posições ainda vazias da população utilizando a parte
       fracionária, da espectativa de cópias de um indivíduo, como percentual
       de probabilidadede de seleção para o mesmo. */
    for ( iCopy = iPosition; iCopy < SIZE_POP; )
    {
        idIndiv = rand(  ) % SIZE_POP;
        fFracCurrent = ( pop[idIndiv].obj/fAverageObj ) - ( int )( pop[idIndiv].obj/fAverageObj );
        iDraw = rand(  ) % 99;
        if ( MAXIMIZATION )/* Verifica se é um problema de maximização ou minimização */
        {
            if (  ( int )( fFracCurrent * 100 ) >=  ( iDraw ) )
            {
                vtPopAux[iPosition] = idIndiv;
                iPosition++;
                iCopy++;
```

```
            }
        }
        else
        {
            if (  ( int )( fFracCurrent * 100 ) <  ( iDraw ) )
            {
                vtPopAux[iPosition] = idIndiv;
                iPosition++;
                iCopy++;
            }
        }
    }
    /* Seleção aleatória de um indivíduo da população temporária */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = vtPopAux[rand(  ) % SIZE_POP];
    }
}
/*******************************************************************************
*Fim: ESTOCÁSTICA POR RESTO SEM REPOSIÇÃO
*******************************************************************************/
```

```
/*************************************************************************
* Título: SELEÇÃO - SELEÇÃO ESTOCÁSTICA SEM REPOSIÇÃO
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Uma população auxiliar será gerada com o número de cópias que um
*            indivíduo tiver direito. Esse número será calculado através da
*            divisão do valor da função objetivo do indivíduo pela média das
*            funções objetivo dos demais indivíduos (fi/fmed).
*            Após este cálculo, o método da roleta será utilizado e a medida que
*            os indivíduos forem sendo selecionados para reprodução os números
*            de cruzamentos permitidos para estes serão decrementados até que
*            cheguem a zero e os indivíduos sejam retirados da roleta.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
*************************************************************************/
#define NORM 10000 /* Normalizador para geração do fitness - Roleta */
void selection( tpPopulation pop, int* pSelected )
{
    int      idIndiv,                /* Identificador do indivíduo da população */
             idNewFather,            /* Indice que referencia os indivíduos selecionados (novos pais)*/
             vtCopy[SIZE_POP];       /* Vetor com o número de cópias de cada indivíduo */
    long int lRandom;                /* Armazena o número randomico gerado */
    float    vtFitness[SIZE_POP],    /* Vetor para armazenamento do fitness */
             fAddFitness         = 0,/* Armazena o somatório do fitness dos indivíduos */
             iTurn               = 0,/* Controla o giro da roleta */
             fAddObj             = 0,/* Soma de todos os valores das funções objetivo */
             fAverageObj;            /* Média dos valores das funções objetivo dos indivíduos da população */
    tpObj    betterObj,              /* Armazena o melhor valor de função objetivo */
             greatValue;             /* Utilizado na criação da roleta caso o problema seja de minimização */

    /* Identificação do maior valor entre as funções objetivo para cálculo do
       fitness */
    betterObj = pop[0].obj;
    greatValue = pop[0].obj;
    for ( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( best( pop[idIndiv].obj , betterObj ) )
        {
            betterObj = pop[idIndiv].obj;
        }
    }

    /* Cálculo e somatório do fitness */
    if ( MAXIMIZATION )
    {
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = ( pop[idIndiv].obj * NORM ) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    else
    {
        for( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( pop[idIndiv].obj > greatValue )
            {
                greatValue = pop[idIndiv].obj;
            }
        }
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = ( ( greatValue - pop[idIndiv].obj ) * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }

    greatValue = pop[0].obj;
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        fAddObj = fAddObj + pop[idIndiv].obj;
        if ( pop[idIndiv].obj > greatValue ) /* Usado quando o problema é de minimização */
        {
            greatValue = pop[idIndiv].obj;
        }
    }

    /* Cálculo da média dos valores das funções objetivo dos indivíduos */
    fAverageObj = fAddObj / SIZE_POP;
```

```
    if ( MAXIMIZATION )
    {
        /* Cálculo do número de cópias, de direito, dos indivíduos */
        for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtCopy[idIndiv] = ceil( pop[idIndiv].obj / fAverageObj );
        }
    }
    else
    {
        for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtCopy[idIndiv] = ceil( ( greatValue - pop[idIndiv].obj ) / fAverageObj );
        }
    }

    /* Seleção dos indivíduos de acordo com o número de pais a serem
       selecionados */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; )
    {
        idIndiv = 0;
        iTurn = vtFitness[idIndiv];
        lRandom = ( rand(  ) * rand(  ) ) % ( long int )fAddFitness;
        /* Giro da roleta */
        while ( iTurn < lRandom )
        {
            idIndiv++;
            iTurn = iTurn + vtFitness[idIndiv];
        }
        if ( vtCopy[idIndiv] > 0 )
        {
            /* Atribuição de um novo indivíuo selecionado à estrutura que será
               retornada */
            pSelected[idNewFather] = idIndiv;
            vtCopy[idIndiv]--;
            idNewFather++;
        }
    }
}

/********************************************************************************
*Fim: SELEÇÃO ESTOCÁSTICA SEM REPOSIÇÃO
********************************************************************************/
```

```
/***************************************************************************
* Título: SELEÇÃO - ESTOCÁSTICA POR RESTO COM REPOSIÇÃO
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Uma população auxiliar será gerada com o número de cópias que um
*            indivíduo tiver direito. Esse número será calculado através da
*            divisão do valor da função objetivo do indivíduo pela média das
*            funções objetivo dos demais indivíduos (fi/fmed).
*            As vagas não preenchidas pelo cálculo anterior, serão preenchidas
*            utilizando o método da roleta. As partes fracionárias, dos cálculos
*            das expectativas de cópias dos indivíduos, serão utilizadas como
*            valores para os setores de uma roleta. A seleção por roleta será
*            executada até que as vagas da população temporária sejam completadas.
*            A seleção aleatória é então aplicada na população temporária para
*            selecionar os indivíduos que participarão do cruzamento.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
***************************************************************************/
#define NORM 10000 /* Normalizador para geração do fitness - Roleta */
void roletaEprr( tpPopulation, int, int* );
void selection ( tpPopulation pop, int* pSelected )
{
    int   idIndiv,                    /* Identificador do indivíduo da população*/
          idNewFather            = 0,/* Indice que referencia os indivíduos selecionados (novos pais)*/
          iPosition              = 0,/* Posição que será ocupada pelo indivíduo na população auxiliar */
          iTotCopies             = 0,/* Total de cópias que um indivíduo tem direito na população auxiliar */
          iCopy,                      /* Variável para controle de iterações de loops */
          iTotSelectRou,              /* Total de indivíduos a ser selecionado pelo método da roleta */
          vtPopAux[SIZE_POP];         /* Vetor que armazena a população auxiliar */
    int*  pSelectedRoulette;          /* Estrutura de dados que recebe o resultado da seleção pelo método da roleta */
    float fPopFrac[SIZE_POP],         /* População formada pela parte fracionária da espectativa de cópias */
          fAddObj                = 0,/* Soma de todos os valores das funções objetivo */
          fAverageObj;                /* Média dos valores das funções objetivo dos indivíduos da população */
    tpObj greatValue;                 /* Utilizado para calcular o número de cópias do indivíduo caso o problema seja de m

    greatValue = pop[0].obj;
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        fAddObj = fAddObj + pop[idIndiv].obj;
        if ( pop[idIndiv].obj > greatValue ) /* Usado quando o problema é de minimização */
        {
            greatValue = pop[idIndiv].obj;
        }
    }

    /* Cálculo da média dos valores das funções objetivo dos indivíduos */
    fAverageObj = fAddObj / SIZE_POP;
    /* Inclusão das cópias dos indivíduos na população auxiliar */
    for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( MAXIMIZATION )
        {
            iTotCopies = ( int )( pop[idIndiv].obj / fAverageObj );
        }
        else
        {
            iTotCopies = ( int )( ( greatValue - pop[idIndiv].obj ) / fAverageObj );
        }
        if ( iTotCopies >= 1 )
        {
            for ( iCopy = 1; iCopy <= iTotCopies; iCopy++ )
            {
                vtPopAux[iPosition] = idIndiv;
                iPosition++;
            }
        }
        fPopFrac[idIndiv] = ( pop[idIndiv].obj/fAverageObj ) - ( int )( pop[idIndiv].obj/fAverageObj );
    }
    /* Preenchimento das posições ainda vazias da população utilizando a parte
       fracionária, da espectativa de cópias de um indivíduo, em um algoritmo do
       tipo roleta */
    iTotSelectRou = SIZE_POP - iPosition;
    pSelectedRoulette = malloc( iTotSelectRou * sizeof( int ) );
    roletaEprr( fPopFrac, iTotSelectRou, pSelectedRoulette );
    /* Inclusão dos indivíduos gerados pelo método da roleta na população
       temporária */
    for ( iCopy = iPosition; iCopy < SIZE_POP; iCopy++ )
    {
        vtPopAux[iPosition] = pSelectedRoulette[idNewFather];
```

```
        iPosition++;
        idNewFather++;
    }
    free( pSelectedRoulette );
    /* Seleção aleatória de um indivíduo da população temporária */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = vtPopAux[rand(  ) % SIZE_POP];
    }
}
/*****************************************************************************
* Título: ROLETA - (Parte da política de seleção ESTOCÁSTICA POR RESTO COM REPOSIÇÃO)
* Versao: 1.0
*  Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Implementação da política de seleção ROLETA na sua forma
*            tradicional, porém recebendo a parte fracionária de uma população e
*            gerando indivíduos para a populaçao temporária da função de seleção.
*            Os valores fracionários das funções objetivo dos indivíduos são
*            utilizados como setores de uma roleta, em seguida a roleta é
*            girada para que os indivíduos sejam selecionados.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          * pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados para participar da população
*                        temporária da função de seleção
*          iTotCompetit- Total de indivíduos a serem selecionados para completar
*                        a população temporária da função de seleção
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através de * pSelected
*****************************************************************************/
void roletaEprr( tpPopulation pop, int iTotCompetit, int* pSelected )
{
    int      idIndiv,                /* Identificador do indivíduo da população*/
             idNewFather;            /* Indice que referencia os indivíduos selecionados (novos pais)*/
    long int lRandom;                /* Armazena o número randomico gerado */
    float    vtFitness[SIZE_POP],    /* Vetor para armazenamento do fitness */
             fAddFitness         = 0,/* Armazena o somatório do fitness dos indivíduos */
             iTurn               = 0;/* Controla o giro da roleta */
    tpObj    betterObj,              /* Armazena o melhor valor de função objetivo */
             greatValue;             /* Utilizado na criação da roleta caso o problema seja de minimização */

    /* Identificação do maior valor entre as funções objetivo para cálculo do
       fitness */
    betterObj = pop[0].obj;
    for ( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
    {
        if ( best( pop[idIndiv].obj , betterObj ) )
        {
            betterObj = pop[idIndiv].obj;
        }
    }

    /* Cálculo e somatório do fitness */
    if ( MAXIMIZATION )
    {
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = (pop[idIndiv].obj * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    else
    {
        for( idIndiv = 1; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( pop[idIndiv].obj > greatValue )
            {
                greatValue = pop[idIndiv].obj;
            }
        }
        for( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            vtFitness[idIndiv] = ( ( greatValue - pop[idIndiv].obj ) * NORM) / betterObj;
            fAddFitness = fAddFitness + vtFitness[idIndiv];
        }
    }
    /* Seleção dos indivíduos de acordo com o número de pais a serem
       selecionados */
    for ( idNewFather = 0; idNewFather < iTotCompetit; idNewFather++ )
    {
        /* Giro da roleta */
```

```
        idIndiv = 0;
        iTurn = vtFitness[idIndiv];
        lRandom = ( ( rand(  ) * rand(  ) ) % ( long int )fAddFitness );
        while ( iTurn < lRandom )
        {
            idIndiv++;
            iTurn = iTurn + vtFitness[idIndiv];
        }
        /* Atribuição de um novo indivíuo selecionado como pai à estrutura que
           será retornada */
        pSelected[idNewFather] = idIndiv;
    }

}

/*******************************************************************************
*Fim: ESTOCÁSTICA POR RESTO COM REPOSIÇÃO
*******************************************************************************/
```

```
/****************************************************************************
* Título: SELEÇÃO - ALEATÓRIO
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 17/12/2005
* Descrição: Nesta função um conjunto de indivíduos é selecionado de forma
*            aleatória.
*
* Entrada: pop       - Estrutura que armazena os cromossomos e funções
*                      objetivos dos indivíduos da população
*          pSelected - Vetor que será retornado com a identificação dos
*                      indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
****************************************************************************/
void selection(tpPopulation pop, int *pSelected)
{
    int idIndiv,      /* Identificador do indivíduo da população*/
        idNewFather;  /* Indice que referencia os indivíduos selecionados (novos pais)*/

    /* Seleção dos indivíduos de acordo com o número de pais a serem
    selecionados */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Atribuição de um novo indivíuo selecionado (pai) à estrutura que será
           retornada */
        pSelected[idNewFather] = rand(  ) % SIZE_POP;
    }
}

/*****************************************************************************
*Fim: ALEATÓRIO
*****************************************************************************/
```

```
/*****************************************************************************
* Título: SELEÇÃO - TORNEIO - Variação: ALEATÓRIO
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 16/03/2007
* Descrição: Nesta função um subconjunto de indivíduos é escolhido de forma
*            aleatória, contudo somente o individuo com maior valor para
*            a função de adequação será selecionado.
*
* Entrada: pop         - Estrutura que armazena os cromossomos e funções
*                        objetivos dos indivíduos da população
*          pSelected - Vetor que será retornado com a identificação dos
*                        indivíduos selecionados
*
* Saída:   a identificação dos indivíduos selecionados é obtida pela função
*          chamadora através da variável pSelected
*****************************************************************************/
#define TOT_COMPETIT 3 /* Total de competidores para participar do torneio */
void selection( tpPopulation pop, int* pSelected )
{
    int   idIndiv,      /* Identificador do indivíduo da população*/
          idNewFather,  /* Indice que referencia os indivíduos selecionados (novos pais)*/
          iCompetitor,  /* Indice que referencia os competidores sorteados*/
          iBetterIndiv; /* Melhor indivíduo do subconjunto sorteado */
    tpObj betterObj;    /* Armazena o melhor valor de função objetivo */

    /* Seleção dos indivíduos de acordo com o número de pais a serem
       selecionados */
    for ( idNewFather = 0; idNewFather < TOT_SELECT; idNewFather++ )
    {
        /* Sorteio de um competidor e definição inicial do mesmo como o melhor */
        idIndiv = rand(  ) % SIZE_POP;
        betterObj = pop[idIndiv].obj;
        iBetterIndiv = idIndiv;
        /* Sorteio dos demais competidores e execução das comparações (torneio) */
        for ( iCompetitor = 1; iCompetitor < TOT_COMPETIT; iCompetitor++ )
        {
            idIndiv = rand(  ) % SIZE_POP;
            if ( best( pop[idIndiv].obj > betterObj ) )
            {
                betterObj = pop[idIndiv].obj;
                iBetterIndiv = idIndiv;
            }
        }
        /* Atribuição de um novo indivíuo selecionado à estrutura que será
           retornada */
        pSelected[idNewFather] = iBetterIndiv;
    }
}
/*****************************************************************************
*Fim: TORNEIO - Variação: ALEATÓRIO
*****************************************************************************/
```

A.3 Código fonte das políticas de evolução da população

```
/****************************************************************************
* Título: TROCA INDIVÍDUOS DA POPULAÇÃO ( Parte da EVOLUÇÃO DA POPULAÇÃO - GAP )
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Troca os piores indivíduos ( pior valor de função objetivo )
*            existentes na população pelos novos indivíduos criados durante os
*            cruzamentos.
*
* Entrada: newPop       - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da nova população
*          pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          iTotCreat    - Número de filhos criados na nova população
*
* Saída:   Uma nova população em pop, com um percentual de indivíduos novos
*          no lugar dos piores indivíduos antigos
****************************************************************************/
void changePop( tpPopulation newPop, tpPopulation pop, int iTotCreat )
{
    int    idCreated, /* Controle do loop dos indivíduos criados */
           idIndiv,   /* Identificador do indivíduo da população*/
           idWorse;   /* Identificador do pior indivíduo */
    tpObj  worseObj;  /* Pior valor de função objetivo */

    for( idCreated=0; idCreated < iTotCreat; idCreated++ )
    {
        worseObj = OBJ_VAL_END;
        for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( (  best( worseObj, pop[idIndiv].obj ) ) && ( pop[idIndiv].obj != OBJ_VAL_INI ) )
            {
                worseObj = pop[idIndiv].obj;
                idWorse = idIndiv;
            }
        }
        pop[idWorse].obj = OBJ_VAL_INI;
    }

    idCreated = 0;
    idIndiv = 0;
    while ( idCreated < iTotCreat )
    {
        if ( pop[idIndiv].obj == OBJ_VAL_INI )
        {
         pop[idIndiv].obj = newPop[idCreated].obj;
         copyChrom( &(pop[idIndiv].chrom), &( newPop[idCreated].chrom ) );
         idCreated++;
        }
        idIndiv++;
    }
}
/****************************************************************************
*FIM: TROCA INDIVÍDUOS DA POPULAÇÃO ( Parte da EVOLUÇÃO DA POPULAÇÃO - GAP )
****************************************************************************/
/****************************************************************************
* Título: EVOLUÇÃO DA POPULAÇÃO - GAP
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Implementação da evolução da população utilizando a política por
*            GAP. Nesta política, apenas um percentual (GAP) de indivíduos da
*            população é substituído pelos seus descendentes, exceto quando o
*            cruzamento não ocorre devido ao critério de probabilidade, fazendo
*            com que os pais permaneçam na nova população.
*
* Entrada: **pPop       - Ponteiro que aponta para outro ponteiro que guarda o
*                         endereço da estrutura que armazena a população.
*                         Obs: Apesar do código não ficar muito legível, este
*                         ponteiro foi definido para que a substituição de uma
*                         população por outra, quando necessária (política SGA),
*                         seja executada simplesmente através da troca de
*                         endereços entre os ponteiros, isto é, com complexidade
*                         O(1).
*
*          *bestIndivid - Ponteiroque aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
*
* Saída:   Uma nova geração de indivíduos é gerada na população
****************************************************************************/
#define GAP 30 /* Porcentagem de novos indivíduos a serem gerados na população */
tpPopulation* populationEvolution( tpPopulation pop, tpIndivid *bestIndivid )
```

```
{
int        idChild,              /* Índice de um filho gerado */
               idNewIndivid = 0,       /* Índice do novo indivíduo criado */
               iProb,                 /* Número randômico gerado para checagem das probabilidades*/
         iTotCreat;            /* Número de filhos a serem criados */
int        selected[TOT_SELECT];  /* vector com a identificação dos indivíduos selecionados */
tpIndivid  children[TOT_CHILDREN];/* vector com os indivíduos gerados em um cruzamento */
tpIndivid  *newPop;               /* Ponteiro para a nova população (válido durante a sua criação) */

    iTotCreat = ( GAP / 100 ) * SIZE_POP;
    newPop = ( tpIndivid* ) malloc ( iTotCreat * sizeof( tpIndivid ) );
    while ( idNewIndivid < iTotCreat )
    {
        /* Aloca a estrutura que armazenará os indivíduos selecionados */
        selection ( pop, selected );
        /* Cruzamento e sua probabilidade */
        iProb=rand(  ) % 100;
        if( ( PROB_CROSS * 100 ) >= iProb )  /*Verifica se o cruzamento deve ser efetuado*/
{
            crossover( pop, selected, children );
        }
        else
        {
            /* Esta função fará com que os pais sejam mantidos para a próxima geração */
            keepFather( pop, selected, children );
        }
        /* Probabilidade de mutação */
        for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
        {
            iProb=rand(  ) % 100;
            if( ( PROB_MUTAT * 100 ) >= iProb )  /*Verifica se a mutação deve ser efetuada*/
    {
                mutation( &children[idChild].chrom );
            }
        }
        for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
        {
            verifyChromChild( &children[idChild].chrom );  /*Checar a viabilidade da solução*/
            calcObj( &children[idChild] );               /*Calcula o valor da função objetivo*/
            bestSolut( &children[idChild], bestIndivid );/*Atualiza a melhor solução*/
        }
        updatePop( newPop, children, &idNewIndivid );    /* Inclusão dos filhos gerados na nova população */
    }

    changePop( newPop, pop, iTotCreat ); /* Substituição dos indivíduos pelos seus descendentes */
    /* Desaloca a estrutura auxiliar utilizada*/
    free( newPop );
    return pop;
}
/*****************************************************************************
*FIM: EVOLUÇÃO DA POPULAÇÃO - GAP
*****************************************************************************/

/****************************************************************************
* Título: ATUALIZAÇÃO DA POPULAÇÃO - GAP (Parte da EVOLUÇÃO DA POPULAÇÃO - GAP)
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que atribui os filhos gerados no cruzamento a estrutura
*            que armazena a nova população (nova geração). É parte integrante da
*            função populationEvolution (EVOLUÇÃO DA POPULAÇÃO - GAP).
*
* Entrada: newPop       - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da nova população
*          *pChildren   - Ponteiro que aponta para a estrutura onde estão os
*                         filhos gerados por um cruzamento
*          *idNewIndivid - Ponteiro que aponta para o local da população onde
*                         deverá ser incluído um novo indivíduo
* Saída:   Um novo conjunto de filhos incluído na nova população(newPop)
****************************************************************************/
void updatePop( tpPopulation newPop, tpIndivid *pChildren, int *idNewIndivid )
{
    int idChild; /* Índice de um filho gerado */
    for( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
    {
        copyIndivid ( &(newPop[*idNewIndivid]), &(pChildren[idChild]));
        (*idNewIndivid)++;
        if ( *idNewIndivid == SIZE_POP )
        {
            break;
        }
    }
}
/*****************************************************************************
*FIM: ATUALIZAÇÃO DA POPULAÇÃO GAP (Parte da EVOLUÇÃO DA POPULAÇÃO - GAP)
*****************************************************************************/
```

```
/******************************************************************************
* Título: EVOLUÇÃO DA POPULAÇÃO - SGA
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Implementação da evolução da população utilizando a política SGA.
*            Na política de evolução SGA, todos os indivíduos da população são
*            substituídos pelos seus descendentes, exceto quando o cruzamento
*            não ocorre devido ao critério de probabilidade.
*
* Entrada: **pPop       - Ponteiro que aponta para outro ponteiro que guarda o
*                         endereço da estrutura que armazena a população.
*                         O edereço armazenado por este ponteiro será, no final
*                         do procedimento, substituído pelo endereço da nova
*                         população.
*                         Obs: Apesar do código não ficar muito legível, este
*                         ponteiro foi definido para que a substituição de uma
*                         população por outra seja executada simplesmente
*                         através da troca de endereços entre os ponteiros,
*                         isto é, com complexidade O(1).
*
*          *bestIndivid - Ponteiroque aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
*
* Saída:   Uma nova geração de indivíduos é gerada na população
******************************************************************************/
tpPopulation* populationEvolution( tpPopulation pop, tpIndivid *bestIndivid )
{
 int        idChild,               /* Índice de um filho gerado */
                idNewIndivid = 0,      /* Índice do novo indivíduo criado */
                iProb;                 /* Número randômico gerado para checagem das probabilidades*/
 int        selected[TOT_SELECT];  /* vector com a identificação dos indivíduos selecionados */
 tpIndivid  children[TOT_CHILDREN];/* vector com os indivíduos gerados em um cruzamento */
 tpIndivid  *newPop;               /* Ponteiro para a nova população (válido durante a sua criação) */

    newPop = ( tpIndivid* ) malloc ( SIZE_POP * sizeof( tpIndivid ) );
    while ( idNewIndivid < SIZE_POP )
 {
 /* Aloca a estrutura que armazenará os indivíduos selecionados */
 selection ( pop, selected );
 /* Cruzamento e sua probabilidade */
 iProb=rand(  ) % 100;
 if( ( PROB_CROSS * 100 ) >= iProb )  /*Verifica se o cruzamento deve ser efetuado*/
 {
  crossover( pop, selected, children );
 }
    else
 {
 /* Esta função fará com que os pais sejam mantidos para a próxima geração */
 keepFather( pop, selected, children );
 }
        /* Probabilidade de mutação */
        for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
        {
           iProb=rand(  ) % 100;
           if( ( PROB_MUTAT * 100 ) >= iProb ) /* Verifica se a mutação deve ser efetuada */
   {
                mutation( &children[idChild].chrom );
           }
        }
        for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
        {
                verifyChromChild( &children[idChild].chrom ); /* Checar a viabilidade da solução */
                calcObj( &children[idChild] );                /* Calcula o valor da função objetivo */
                bestSolut( &children[idChild], bestIndivid ); /* Atualiza a melhor solução */
        }
 updatePop( newPop, children, &idNewIndivid ); /* Atualiza a população */

 }
   deallocPopulation( pop );  /* Desaloca a estrutura auxiliar utilizada */
   return newPop;
}
/******************************************************************************
*FIM: EVOLUÇÃO DA POPULAÇÃO - SGA
******************************************************************************/

/******************************************************************************
* Título: ATUALIZAÇÃO DA POPULAÇÃO - SGA (Parte da EVOLUÇÃO DA POPULAÇÃO - SGA)
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que atribui os filhos gerados no cruzamento a estrutura
*            que armazena a nova população (nova geração). É parte integrante da
*            função populationEvolution (EVOLUÇÃO DA POPULAÇÃO - SGA).
*
* Entrada: newPop       - Estrutura que armazena os cromossomos e funções
```

```
*                         objetivos dos indivíduos da nova população
*          *pChildren   - Ponteiro que aponta para a estrutura onde estão os
*                         filhos gerados por um cruzamento
*          *idNewIndivid - Ponteiro que aponta para o local da população onde
*                         deverá ser incluído um novo indivíduo
* Saída:   Um novo conjunto de filhos incluído na nova população(newPop)
******************************************************************************/
void updatePop( tpPopulation newPop, tpIndivid *pChildren, int *idNewIndivid )
{
    int idChild; /* Índice de um filho gerado */
    for( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
    {
        copyIndivid ( &(newPop[*idNewIndivid]), &(pChildren[idChild]));
        (*idNewIndivid)++;
        if ( *idNewIndivid == SIZE_POP )
        {
            break;
        }
    }
}
/*******************************************************************************
*FIM: ATUALIZAÇÃO DA POPULAÇÃO SGA (Parte da EVOLUÇÃO DA POPULAÇÃO - SGA)
*******************************************************************************/
```

```
/***************************************************************************
* Título: EVOLUÇÃO DA POPULAÇÃO - SSGA
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Implementação da evolução da população utilizando a política SSGA.
*            Na política de evolução SSGA, apenas um indivíduo é gerado a cada
*            geração, exceto quando o cruzamento não ocorre devido ao critério
*            de probabilidade, fazendo com que nenhum indivíduo seja gerado
*            naquela geração.
*
* Entrada: **pPop       - Ponteiro que aponta para outro ponteiro que guarda o
*                         endereço da estrutura que armazena a população.
*                         Obs: Apesar do código não ficar muito legível, este
*                         ponteiro foi definido para que a substituição de uma
*                         população por outra, quando necessária (política SGA),
*                         seja executada simplesmente através da troca de
*                         endereços entre os ponteiros, isto é, com complexidade
*                         O(1).
*
*          *bestIndivid - Ponteiroque aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
*
* Saída:   Uma nova geração de indivíduos é gerada na população
***************************************************************************/
tpPopulation* populationEvolution( tpPopulation pop, tpIndivid *bestIndivid )
{
int        idChild,              /* Índice de um filho gerado */
               idNewIndivid=0,        /* Índice do novo indivíduo criado */
           iProb;                /* Número randômico gerado para checagem das probabilidades*/
int        selected[TOT_SELECT];  /* vector com a identificação dos indivíduos selecionados */
tpIndivid  children[TOT_CHILDREN];/* vector com os indivíduos gerados em um cruzamento */

/* Aloca a estrutura que armazenará os indivíduos selecionados */
selection ( pop, selected );
/* Cruzamento e sua probabilidade */
iProb=rand(  ) % 100;
if((PROB_CROSS * 100) >= iProb)  /*Verifica se o cruzamento deve ser efetuado*/
{
        crossover( pop, selected, children );
}
else
{
        /* Esta função fará com que os pais sejam mantidos para a próxima geração */
keepFather( pop, selected, children );
    }
    /* Probabilidade de mutação */
    for( idChild=0; idChild<TOT_CHILDREN; idChild++ )
    {
        iProb=rand(  ) % 100;
        if( ( PROB_MUTAT * 100 ) >= iProb )  /*Verifica se a mutação deve ser efetuada*/
    {
            mutation( &children[idChild].chrom );
        }
    }
    for( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
    {
        verifyChildren( &children[idChild].chrom );   /*Checar a viabilidade da solução*/
        calcObj( &children[idChild] );               /*Calcula o valor da função objetivo*/
        bestSolut( &children[idChild], bestIndivid ); /*Atualiza a melhor solução*/
    }
updatePop( pop, children, &index ); /*Atualiza a população*/
return pop;
}
/****************************************************************************
*FIM: EVOLUÇÃO DA POPULAÇÃO - SSGA
****************************************************************************/

/***************************************************************************
* Título: ATUALIZAÇÃO DA POPULAÇÃO- SSGA (Parte da EVOLUÇÃO DA POPULAÇÃO - SSGA)
* Versao: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que atribui os filhos gerados no cruzamento a estrutura
*            que armazena a população. Os novos índivíduos substituem os piores
*            índivíduos ( pior valor de função objetivo ) da população.
*            É parte integrante da função populationEvolution (EVOLUÇÃO DA
*            POPULAÇÃO - SSGA).
*
* Entrada: newPop       - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da nova população
*          *pChildren   - Ponteiro que aponta para a estrutura onde estão os
*                         filhos gerados por um cruzamento
*          *pSelected   - Ponteiro que aponta para a estrutura com a
*                         identificação dos indivíduos selecionados pela
*                         política de seleção
```

```
* Saída:   Um novo conjunto de filhos, gerado por um cruzamento, incluído na
*          população
*************************************************************************/
void updatePop( tpPopulation pop, tpIndivid *pChildren, int *pSelected )
{

    int     idChild,   /* Índice de um filho gerado */
            idIndiv,   /* Identificador do indivíduo da população */
            idWorse;   /* Identificador do pior indivíduo */
    tpObj   worseObj;  /* Pior valor de função objetivo */

    for ( idChild = 0; idChild < TOT_CHILDREN; idChild++ )
    {
        worseObj = OBJ_VAL_END;
        for ( idIndiv = 0; idIndiv < SIZE_POP; idIndiv++ )
        {
            if ( (  best( worseObj, pop[idIndiv].obj ) ) && ( pop[idIndiv].obj != OBJ_VAL_INI ) )
            {
                worseObj = pop[idIndiv].obj;
                idWorse = idIndiv;
            }
        }
        pop[idWorse].obj = OBJ_VAL_INI;
    }
    idChild = 0;
    idIndiv = 0;
    while ( idChild < TOT_CHILDREN )
    {
        if ( pop[idIndiv].obj == OBJ_VAL_INI )
        {
            pop[idIndiv].obj = children[idChild].obj;
            copyChrom( &(pop[idIndiv].chrom), &( children[idChild].chrom ) );
            idChild++;
        }
        idIndiv++;
    }
}
/*****************************************************************************
*FIM: ATUALIZAÇÃO DA POPULAÇÃO SSGA (Parte da EVOLUÇÃO DA POPULAÇÃO - SSGA)
*****************************************************************************/
```

A.4 Código fonte dos critérios de parada

```
/******************************************************************************
* Título: CRITÉRIO DE PARADA - TEMPO PREDEFINIDO DE COMPUTAÇÃO
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que define como critério de parada do algoritmo genético um
*            tempo predefinido de processamento.
*
* Entrada: pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
******************************************************************************/
tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid )
{
    long int iMaxTime;
    clock_t time1,time2;

    printf("Entre com o tempo de processamento (segundos):");
    scanf("%d",&iMaxTime);
    time1=clock();
    do
    {
        pop = populationEvolution ( pop, bestIndivid );
        time2=clock();
        fElapsedTime = (float) (time2-time1) / CLOCKS_PER_SEC;
    }
    while ( fElapsedTime <= iMaxTime );
    return pop;
}
/******************************************************************************
*FIM: CRITÉRIO DE PARADA - TEMPO PREDEFINIDO DE COMPUTAÇÃO
******************************************************************************/
```

```
/*****************************************************************************
* Título: CRITÉRIO DE PARADA-MELHORA MÍNIMA EM UM NÚMERO DETERMINADO DE GERAÇÕES
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que define como critério de parada do algoritmo genético uma
*            melhora mínima obrigatória para a função objetivo, durante um certo
*            número de gerações.
*
* Entrada: pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
*****************************************************************************/
tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid )
{
    int   iGenerat    = 0, /* Identificador da geração */
          iMaxGen;         /* Número máximo de gerações após cada melhora */
    tpObj improve,         /* Melhora mínima para continuar o processamento */
          bestObjPrev = 0; /* Melhor valor de função objetivo do conjunto de iterações anterior*/

    printf("Entre com o número máximo de gerações por melhora:");
    scanf("%d",&iMaxGen);
    printf("Entre com o valor da melhora mínima:");
    scanf("%f",&improve);

    do
    {
        iGenerat++;
        pop = populationEvolution ( pop, bestIndivid );
        if ( ((*bestIndivid).obj - bestObjPrev) >= improve)
        {
            iGenerat = 0;
            bestObjPrev = (*bestIndivid).obj;
        }
    }
    while ( iGenerat <= iMaxGen );
    return pop;
}
/*****************************************************************************
*FIM: CRITÉRIO DE PARADA - MELHORA MÍNIMA EM UM NÚMERO DETERMINADO DE GERAÇÕES
*****************************************************************************/
```

```
/******************************************************************************
* Título: CRITÉRIO DE PARADA - NÚMERO MÁXIMO DE GERAÇÕES
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que define como critério de parada do algoritmo genético um
*            número máximo de gerações definido pelo usuário.
*
* Entrada: pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
******************************************************************************/
#define MAX_GEN 1000    /* Número máximo de gerações para o algoritmo */
tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid )
{
   int  iGenerat;/* Identificador da geração */

   iGenerat = 0;
   while ( iGenerat < MAX_GEN ) /*Controla o número total de gerações*/
   {
      pop = populationEvolution ( pop, bestIndivid );
      iGenerat++;
   }
   return pop;
}
/******************************************************************************
*FIM: CRITÉRIO DE PARADA - NÚMERO MÁXIMO DE GERAÇÕES
******************************************************************************/
```

```
/******************************************************************************
* Título: CRITÉRIO DE PARADA - CONVERGÊNCIA DA POPULAÇÃO
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que define como critério de parada do algoritmo genético um
*            certo valor mínimo de heterogeneidade dos indivíduos da população
*            ( avaliado em relação aos valores das funções objetivo ).
*
* Entrada: pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
******************************************************************************/
tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid ) /* Ainda não foi testada */
{
    float fAverage,
          fConverg;
    int   idInd;

    printf("Entre com o valor para a convergência:");
    scanf("%d",&fConverg);
    do
    {
        pop = populationEvolution ( pop, bestIndivid );
        fAverage = 0;
        for ( idInd = 0; idInd < SIZE_POP; idInd++ )
        {
            fAverage = fAverage + pop[idInd].obj;
        }
        fAverage = fAverage / SIZE_POP;
    }
    while ( ( (*bestIndivid).obj - fAverage )> fConverg );
    return pop;
}
/******************************************************************************
*FIM: CRITÉRIO DE PARADA - CONVERGÊNCIA DA POPULAÇÃO
******************************************************************************/
```

```
/*****************************************************************************
* Título: CRITÉRIO DE PARADA - VALOR ALVO
* Versão: 1.0
* Autor: Fábio Duncan de Souza
* Data: 02/03/2007
* Descrição: Função que define como critério de parada do algoritmo genético um
*            certo valor de função objetivo a ser atingido.
*
* Entrada: pop          - Estrutura que armazena os cromossomos e funções
*                         objetivos dos indivíduos da população
*          *bestIndivid - Ponteiro que aponta para a estrutura do melhor
*                         indivíduo gerado pelo algoritmo genético
* Saída:   A melhor solução para o problema em *bestIndivid (indivíduo melhor
*          adaptado)
*****************************************************************************/
tpPopulation* stopCriterion ( tpPopulation pop, tpIndivid *bestIndivid )
{
    tpObj target; /* Valor a ser atingido para finalizar o algoritmo */

    printf("Entre com o valor alvo:");
    scanf("%f",&target);
    do
    {
        pop = populationEvolution ( pop, bestIndivid );
    }
    while ( (*bestIndivid).obj < target );
    return pop;
}
/*****************************************************************************
*FIM: CRITÉRIO DE PARADA - VALOR ALVO
*****************************************************************************/
```