GENETICFRAME – FRAMEWORK PARA ALGORITMOS GENÉTICOS

Antônio Almeida de Barros Junior
Cientec - Consultoria e Desenvolvimento de Sistemas Ltda.
Av. P. H. Rolfs 305/20 – Viçosa – MG – 36570-000
antoniojr@cientec.net

Alessandro de Freitas Teixeira
Cientec - Consultoria e Desenvolvimento de Sistemas Ltda.
Av. P. H. Rolfs 305/20 – Viçosa – MG – 36570-000
alessandro@cientec.net

Paulo Márcio de Freitas
Cientec - Consultoria e Desenvolvimento de Sistemas Ltda.
Av. P. H. Rolfs 305/20 – Viçosa – MG – 36570-000
paulo@cientec.net

Resumo

Os Algoritmos Genéticos são muito empregados em técnicas de busca e otimização e têm como natureza manter uma analogia à genética natural. Devido a esta característica, sua implementação mantém estruturas em comum nos mais diversos tipos de problemas. Com o objetivo de promover a padronização e reusabilidade destas características, o presente trabalho propõe um *framework* para Algoritmos Evolutivos denominado GeneticFrame. Desenvolvido em uma estrutura totalmente orientada a objetos, o *framework* explora esta característica para facilitar a sua extensão para algoritmos genéticos e problemas específicos, sejam eles uni ou multiobjetivos. Neste trabalho é apresentado um exemplo da utilização do GeneticFrame para uma implementação de um algoritmo genético multiobjetivo. Nas implementações realizadas mostrou-se eficaz no que diz respeito à extensibidade e reusabilidade do código no desenvolvimento de aplicações, tanto uniobijetivo, como multiobjetivo. Com isto promoveu a simplicidade e a facilidade de manutenção no código das aplicações desenvolvidas.

Palavras-Chaves: Framework; Algoritmo Genético; Otimização.

Abstract

Genetic Algorithms are very employed in searching and optimization techniques, and they mainly maintain an analogy to the natural genetics. Due to this characteristic, its implementation maintains similar structures in most types of problems. With the objective of promoting the standardization and reusability of these characteristics, the present work proposes a framework for Evolutionary Algorithms denominated GeneticFrame. Developed in an object oriented structure, the framework explores this characteristic to facilitate its extension for genetic algorithms and specific problems, be uni or multiobjectives. This work presents an example of use of GeneticFrame for an algorithm genetic multiobjetive implementation. In the implementations made, it proved to be effective in what it concerns the code extensibility and reusability in the applications development with uniobjective or multiobjetives ones. By this, it promoted simplicity and easiness in maintaining the code of the applications developed.

Keywords: Framework, Genetic Algorithm; Optimization.

1. INTRODUÇÃO

Algoritmos Genéticos (GAs - Genetic Algorithms) constituem uma técnica de busca e otimização, altamente paralela, inspirada no princípio Darwiniano de seleção natural e reprodução genética [1].

Para manter a analogia são usados nos sistemas artificiais os termos pertinentes à genética natural. Desta forma, um indivíduo ou estrutura corresponde a uma concatenação de variáveis ou *cadeias de caracteres* (cromossomos), onde cada caractere (gene), encontra-se numa dada posição (lócus) e com seu valor determinado (alelo). Um sinônimo de indivíduo em genética natural é o genótipo e a sua estrutura decodificada é o fenótipo. Em outras palavras o genótipo, em sistemas artificiais significa um conjunto de parâmetros, ou um ponto solução no espaço de procura.[2].

Detalhes a respeito da analogia entre Algoritmos Genéticos e o sistema natural estão representados no Quadro 1:

Quadro 1 – Analogia entre Sistema Natural e GAs [3].

Natureza	**Algoritmos Genéticos**
Cromossoma	Palavra binária, vetor, etc.
Gene	Característica do problema
Alelo	Valor da característica
Loco	Posição na palavra, vetor
Genótipo	Estrutura
Fenótipo	Estrutura submetida ao problema
Geração	Ciclo

Os Algoritmos Genéticos utilizam elementos como a sobrevivência dos mais aptos e a troca de informações genética de uma forma estruturada [4]. Embora o Algoritmo Genético use um método heurístico e probabilístico para obter os novos elementos, ele não pode ser considerado uma simples busca aleatória, uma vez que explora inteligentemente as informações disponíveis de forma a buscar novos indivíduos [5].

Devido a estas características em comum para todo Algoritmo Genético, foi possível a implementação do *framework* proposto neste artigo.

Um “Framework” é o projeto de um conjunto de objetos que colaboram entre si para execução de um conjunto de responsabilidades. Um *framework* reusa análise, projeto e código. Ele reusa análise porque descreve os tipos de objetos importantes e como um problema maior pode ser dividido em problemas menores. Ele reusa projeto porque contém algoritmos abstratos e descreve a interface que o programador deve implementar e as restrições a serem satisfeitas pela implementação. Ele reusa código porque torna mais fácil desenvolver uma biblioteca de componentes compatíveis e porque a implementação de novos componentes pode herdar grande parte de seu código das super-classes abstratas. Apesar de todos os tipos de reuso serem importantes, o reuso de análise e de projeto são os que mais compensam a longo prazo [6]. Para Coad [7] um *framework* é definido como um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas.

Os benefícios do uso de *framework*, além do reuso, da facilidade de manutenção e da inversão de controle, são a capacidade de extensão e a modularidade. A modularidade é atingida quando os detalhes de implementação são encapsulados pelo *framework* em classes fixas. Já sua extensibilidade se dá devido ao *framework* disponibilizar “esqueletos” de métodos, permitindo que as classes possam ser estendidas. A extensão de um *framework* é uma característica fundamental para economizar tempo de programação [8].

Os *frameworks* têm algumas características de reuso de sistemas “caixa branca”, uma vez que certo grau de conhecimento sobre sua estrutura é necessário para conseguir construir uma aplicação. Um grande problema que esse tipo de reuso traz é que, à medida que a

complexidade da estrutura aumenta, maior deve ser o conhecimento sobre ela para se desenvolver uma aplicação. Além disso, o desenvolvimento de sistemas baseado em reuso de *frameworks* acarreta certa perda de desempenho [9].

O uso deste tipo de tecnologia pode proporciona como resultado, a construção de aplicações mais rapidamente, como também aplicações que tenham estruturas similares. Elas são mais fáceis de manter e parecem mais consistentes para seus usuários [10].

Este trabalho tem como objetivo, propor um modelo de *framework* para Algoritmos Evolutivos denominado GeneticFrame. O modelo apresentado se adequa bem aos sistemas de otimização que utilizam a metaheurística de Algoritmos Genéticos. Uma das características predominantes neste *framework* é o fato do mesmo estar preparado para Algoritmos Genéticos na solução de problemas uniobjetivo e multiobjetivo.

2. GENETICFRAME

Com base nas idéias anteriormente apresentadas, desenvolveu-se o *framework* GeneticFrame com o objetivo de proporcionar a padronização de sistemas que utilizam a metaheurística de algoritmos genéticos. A Figura 1 apresenta o diagrama de classes em UML da arquitetura geral do *framework* proposto.

Sua implementação foi realizada utilizando-se a linguagem Object Pascal e a ferramenta Borland Delphi 7.

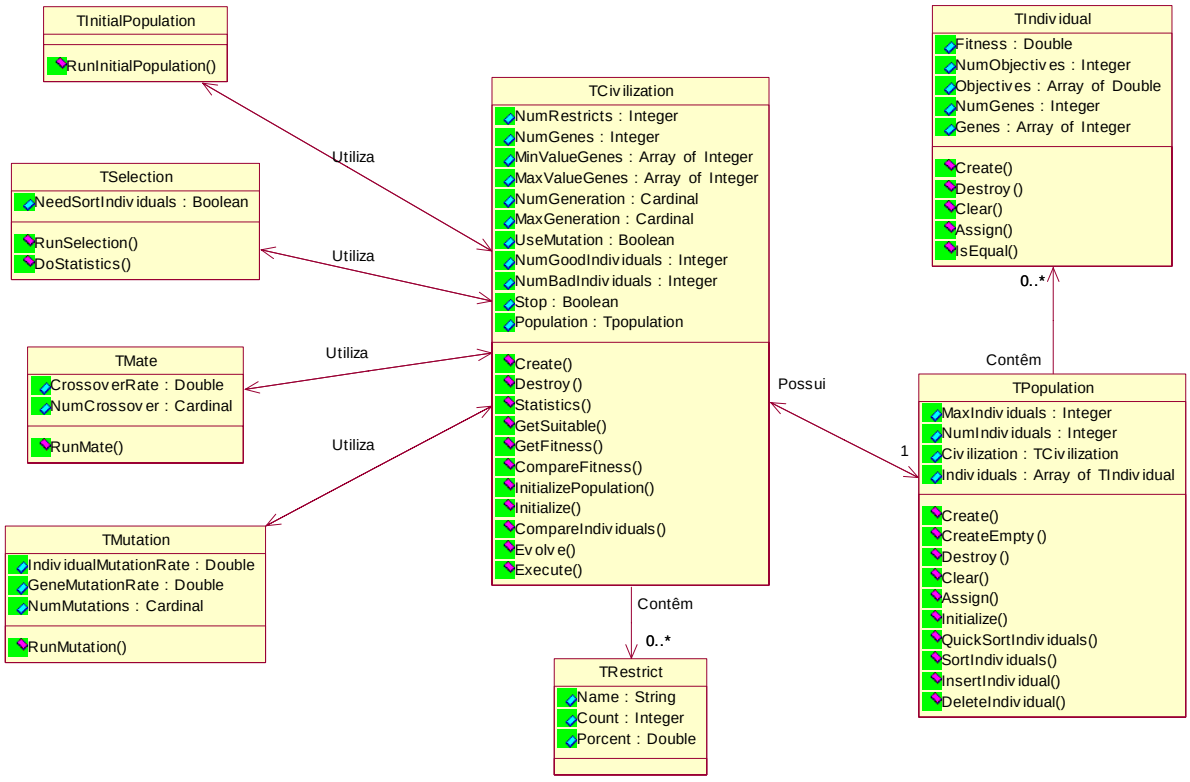

Figura 1 – Arquitetura geral do *framework* GeneticFrame.

As descrições das classes são mostradas nas seções a seguir.

2.1. TCIVILIZATION

TCivilization é a principal classe do *framework* e sua função está diretamente relacionada ao processo de evolução da população. Esta classe possui os principais

parâmetros de controle do algoritmo genético e é responsável pela execução do processo evolutivo. É uma classe abstrata e serve de interface para os principais atributos e métodos dos algoritmos genéticos. Possui também os métodos de controle do fluxo principal de evolução. Para a solução de um problema real utilizando este *framework* é necessário criar uma subclasse desta e definir alguns métodos concretos. Caso necessário pode-se adicionar novos métodos relacionados à solução do problema específico que será tratado.

A seguir estão descritos os principais atributos da classe *TCivilization*:

- *Population*: atributo do tipo *TPopulation* que contém a população que está sofrendo a evolução.
- *NumGenes*: armazena o número de genes existentes no cromossomo;
- *MinValueGenes*: vetor que armazena o valor mínimo para o alelo de cada gene;
- *MaxValueGenes*: vetor que armazena o valor máximo para o alelo de cada gene;
- *MaxGeneration*: número máximo de gerações a ser atingido;
- *NumGeneration*: armazena o número da geração durante o processamento. Pode chegar ao número máximo (*MaxGeneration*);
- *UseMutation*: informa se o operador de mutação deve ou não ser utilizado;
- *NumGoodIndividuals*: valor estatístico que armazena o número de indivíduos viáveis gerados;
- *NumBadIndividuals*: valor estatístico que armazena o número de indivíduos inviáveis gerados;
- *Stop*: *flag* que sinaliza a interrupção na execução do processo evolutivo;
- *Restricts*: vetor de objetos do tipo *TRestrict* que contém as restrições do modelo de otimização. O atendimento a estas restrições é que determinam a viabilidade ou não de um indivíduo;
- *NumRestricts*: número de restrições do vetor de restrições.

A seguir estão descritos os principais métodos da classe *TCivilization*:

- *Execute()*: Este é o método que dispara o processo evolutivo. Seu funcionamento consiste basicamente em gerar a população inicial e entrar no processo iterativo de evolução da população, como mostra o algoritmo da Figura 2. No início e no final de cada iteração do processo evolutivo o *framework* possibilita a execução de eventos que permitem a extensão das funcionalidades deste método. Esta é uma característica importante do *framework* que o deixa mais extensível, permitindo a mudança ou complementação do comportamento do método e facilitando sua utilização na solução de problemas reais. Para realizar esta tarefa é utilizado o conceito de ponteiros de métodos da linguagem ObjectPascal. Métodos podem ser atribuídos aos eventos *FOnBeforeEvolve()* e *FOnNewPopulation()* em tempo de execução.

```
procedure TCivilization.Execute();
begin
    Initialize;
    InitializePopulation;
    do
        if Assigned(FOnBeforeEvolve) then
            FOnBeforeEvolve(Self);
        Evolve;
        Statistics;
        if Assigned(FOnNewPopulation) then
            FOnNewPopulation(Self);
    while NumGeneration >= MaxGeneration;
end;
```

Figura 2 – Método *Execute()* da Classe *TCivilization.*

- *Initialize()*: Neste método são zerados alguns atributos da classe para iniciar o processamento.

- *InitializePopulation()*: Este método é responsável por criar a população inicial. Sua funcionalidade básica é executar o método *RunInitialPopulation()* da classe *TInitialPopulation*, detalhada mais adiante.

- *Evolve()*: Após a criação da população inicial, é iniciado o processo iterativo de evolução das gerações. O método *Evolve()* é o responsável pela evolução de uma geração de indivíduos, ou seja, executa os métodos que permitem a evolução da população, como mostra o algoritmo da Figura 3. Neste método são executadas as operações de seleção, cruzamento e mutação, que caracterizam o algoritmo genético.

```
procedure TCivilization.Evolve;
    auxPopulation: TPopulation;
    auxIndividual, aIndividual, bIndividual: TIndividual;

    procedure ProcessIndividual(var Pop: TPopulation; var NewIndividual: TIndividual);
    begin
        if UseMutation and Happend(Mutation.IndividualMutationRate) then
            Mutation.RunMutation(NewIndividual);
        NewIndividual.Fitness := GetFitness(NewIndividual); //Calcula o fitness

        if GetSuitable(NewIndividual.Genes) then
            Pop.InsertIndividual(NewIndividual)
        else
            Pop.DeleteIndividual(NewIndividual);
    end;

begin
    auxPopulation := TPopulation.Create(Self);

    while auxPopulation.NumIndividuals < auxPopulation.MaxIndividuals do
    begin
        auxIndividual := Selection.RunSelection;
        aIndividual := TIndividual.Create(auxIndividual); //Faz uma cópia do indivíduo
        auxIndividual := Selection.RunSelection;
        bIndividual := TIndividual.Create(auxIndividual); //Faz uma cópia do indivíduo

        if Happend(Mate.CrossoverRate) then
            Mate.RunMate(aIndividual, bIndividual);

        ProcessIndividual(auxPopulation, aIndividual);
        if auxPopulation.NumIndividuals < auxPopulation.MaxIndividuals then
            ProcessIndividual(auxPopulation, bIndividual);

        if Stop then
            raise Exception.Create ('Processo interrompido pelo usuário.');
    end;
    Population.Free;
    Population := auxPopulation;
    NumGeneration := NumGeneration + 1;
```

fim;

Figura 3 – Método *Evolve()* da Classe *TCivilization.*

- *Statistics()*: Responsável por atualizar os atributos referentes às estatísticas do processo evolucionário.

- *GetSuitable()*: Responsável por verificar a viabilidade do indivíduo, ou seja, se ele atende as restrições do problema. Este método faz a consulta ao vetor de restrições *Restricts*, aqui representadas por objetos do tipo *TRestrict* (detalhado mais à frente), passando como parâmetro o indivíduo a ser testado. Esta é outra característica importante do *framework* proposto, onde uma estrutura de dados genérica foi desenvolvida para armazenar as funções de verificação das restrições do problema. Caso o indivíduo atenda as restrições previstas o método *GetSuitable()* retornará "Verdadeiro" e caso o indivíduo não atenda às restrições, o método retornará "Falso".

- *GetFitness()*: Retorna o valor do *fitness* de um determinado indivíduo, passado via parâmetro. O método para cálculo do *fitness* deve ser também uma estrutura genérica, pois o *fitness* é um atributo que varia com o problema que está sendo estudado. Neste *framework* a forma proposta para atender esta condição foi criar um atributo chamado *CalculateFitness()* do tipo ponteiro de método. Em tempo de execução deve ser atribuído, então, um método que calcule o *fitness* do indivíduo.

- *CompareIndividuals()*: Recebe como parâmetro dois indivíduos e deve retornar qual deles é melhor, levando em consideração o modo de convergência (minimização ou maximização) e o(s) objetivo(s) a ser(em) considerado(s) na solução do problema. Este método é abstrato e deve ser redefinido na subclasse criada a partir da *TCivilization*, pois, dependendo das características do algoritmo genético que está sendo implementado a forma de definição de qual indivíduo é melhor pode mudar. Este é um método muito utilizado nos operadores de seleção de indivíduos para cruzamento.

2.2. TPOPULATION

Uma população é a composição de um conjunto de indivíduos. Basicamente a classe *TPopulation* é composta de um vetor de objetos da classe *TIndividual* e alguns métodos de manipulação destes indivíduos. No *framework* proposto uma característica importante é a existência de um apontador para a classe *TCivilization* que a criou, permitindo que a classe *TPopulation* acesse informações e métodos da civilização.

A classe *TPopulation* possui os seguintes atributos:

- *MaxIndividuals*: armazena o número máximo de indivíduos da população;

- *NumIndividuals*: armazena o número de indivíduos que a população possui, podendo chegar a um número máximo de indivíduos (*MaxIndividuals*);

- *Individuals*: é o vetor que armazena os indivíduos da população;

- *Civilization*: apontador para a classe *TCivilization* que instanciou a população. Esta referência à classe *TCivilization* é importante para as operações realizadas nos métodos da classe *TPopulation*.

Os métodos da classe *TPopulation* são:

- *Clear()*: Apaga todos os indivíduos da população.

- *Assign()*: Realiza a cópia da população passada por parâmetro para a população corrente.

- *Initialize()*: Responsável por zerar os atributos da classe *TPopulation*.

- *SortIndividual()*: Realiza a ordenação dos indivíduos da população.

- *InsertIndividual()*: Adiciona um novo indivíduo à população.

- *DeleteIndividual()*: Exclui um indivíduo da população.

2.3. TINDIVIDUAL

A classe *TIndividual* representa a menor granularidade do *framework*. Cada um dos indivíduos de uma população representa uma possível solução para o problema, ou seja, um ponto no espaço de soluções.

A classe *TIndividual* possui os seguintes atributos:
- *Fitness*: armazena o valor do *fitness* de um indivíduo.
- *NumObjectives*: armazena o número de objetivos considerados, no caso de problemas multiobjetivos.
- *Objectives*: vetor de *NumObjectives* posições, onde cada posição armazena o valor do indivíduo para um objetivo.
- *NumGenes*: armazena o número de genes que um indivíduo possui.
- *Genes*: vetor de *NumGenes* posições, onde cada posição armazena o valor do gene ou alelo.
- *Population*: apontador para a classe *TPopulation* que instanciou o indivíduo. Esta referência à classe *TPopulation* pode ser importante para as operações realizadas nos métodos da classe *TIndividual*.

Além dos atributos, a classe *TIndividual* contêm os seguintes métodos principais:
- *Clear()*: Apaga todos os genes e objetivos do indivíduo.
- *Assign()*: Realiza a cópia do indivíduo passado por parâmetro para o indivíduo corrente.
- *IsEqual()*: Compara, gene a gene, se dois indivíduo são iguais.

2.4. TINITIALPOPULATION

É uma classe abstrata que serve de interface para a utilização de diferentes métodos de geração da população inicial. Portanto, para utilizá-la é necessário criar uma subclasse para um método específico de geração dos indivíduos da população inicial. A classe não dispõe de atributos e contêm um único método abstrato *RunInitialPopulation()* que deve ser redefinido na subclasse implementada.

2.5. TMATE

Esta é a classe que implementa o operador de cruzamento de indivíduos (*crossover*). É uma classe abstrata que serve de interface para a definição de diferentes métodos de cruzamento. Para cada tipo de cruzamento, deve-se definir uma nova subclasse a partir de *TMate*. O tipo de cruzamento determina a forma como se procederá a troca de informações entre indivíduos de cromossomos selecionados.

Os atributos disponíveis na classe são:
- *CrossoverRate*: armazena a taxa de cruzamento entre indivíduos;
- *NumCrossover*: é um valor estatístico do número de cruzamentos já realizados;

RunMate() é o único método disponível na classe *TMate*. Por ser um método abstrato no *framework* deve ser redefinido na subclasse.

2.6. TMUTATION

A classe *TMutation* implementa o operador de mutação dos indivíduos. Por ser uma classe abstrata, serve apenas de interface e deve ser derivada para a definição de métodos específicos de mutação.

Os atributos disponíveis na classe são:

- *IndividualMutationRate*: define o percentual de indivíduos da população que sofrerão mutação.
- *GeneMutationRate*: define o percentual de genes do indivíduo que sofrerão mutação.
- *NumMutations*: valor estatístico que armazena o número de mutações realizadas;

RunMutation() é o único método disponível na classe. Por ser um método abstrato no *framework* deve ser redefinido na subclasse.

2.7. TSELECTION

A classe *TSelection* implementa o operador de seleção de indivíduos para cruzamento.

O único atributo disponível na classe é *NeedSortIndividuals* que define se o método de seleção implementado necessita da ordenação dos indivíduos da população.

O método *RunSelection()* é abstrato e deve ser redefinido na subclasse. O método *DoStatistics* é um método abstrato que deve ser implementado na subclasse caso o método de seleção utilize de variáveis estatísticas calculadas durante o processo de evolução da civilização para seu processamento. Este método é executado no método *DoStatistics* da classe *TCivilization*.

2.8. TRESTRICT

A classe *TRestrict* representa uma restrição imposta ao problema. Esta é uma estrutura de dados genérica que permite ao *framework* abstrair as funções que calculam as restrições do problema, ou seja, que analisam se um indivíduo (possível solução no espaço de soluções) é viável (atende às restrições do problema).

Os atributos da classe *TRestrict* são:

- *Name*: nome descritivo da restrição imposta ao problema;
- *Count*: valor estatístico que armazena o número de indivíduos que não atenderam à restrição.
- Porcent: valor estatístico que armazena o percentual de indivíduos que não atenderam à restrição.

O atributo *VerifySuitable()* é um ponteiro de método que permite a atribuição, em tempo de execução, do método que calcula a restrição.

3. APLICAÇÃO DO FRAMEWORK

Para exemplificar a utilização do *framework* GeneticFrame, nesta seção é mostrada uma implementação da metaheurística de algoritmos genéticos para problemas multiobjetivo SPEA - *Strength Pareto Evolutionary Algorithm* [11]. Neste contexto foi criada a classe *TSpea* que é derivada da classe *TCivilization* pertencente ao *framework* GeneticFrame (Figura 4). Nesta nova classe podem ser observados os atributos e métodos específicos do SPEA, bem como a sobrecarga de alguns métodos da classe *TCivilization*. O atributo *NonDominatedSet* é uma característica do algoritmo SPEA que mantêm uma população de indivíduos pareto ótimos ao longo da evolução. Os métodos específicos, como é o caso do método *SPEA()*, implementa as características do algoritmo complementando as funcionalidades da classe *TCivilization*.

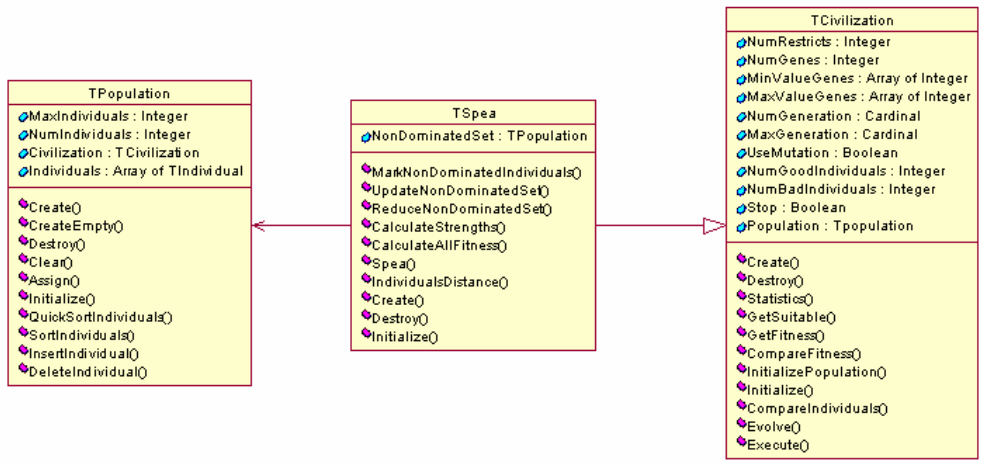

Figura 4 – Classe *TSpea* na solução de problemas multiobjetivo.

Da mesma forma que foi criada a subclasse *TSpea*, que estende o algoritmo para a metaheurística SPEA de otimização multiobjetivo, também foram criadas extensões de outras classes importantes do *framework*.

Para a criação da população inicial foi implementadas três novas classes (*TMutationIdealInitialPopulation*, *TSimilarIdealInitialPopulation* e *TRandomInitialPopulation*) que estendem a classe abstrata *TInitialPopulation*, como mostra a Figura 5. Todas as classes contêm o método *RunInitialPopulation()* que sobrepõe o da classe herdada, cada um contendo uma técnica diferente de geração da população inicial.

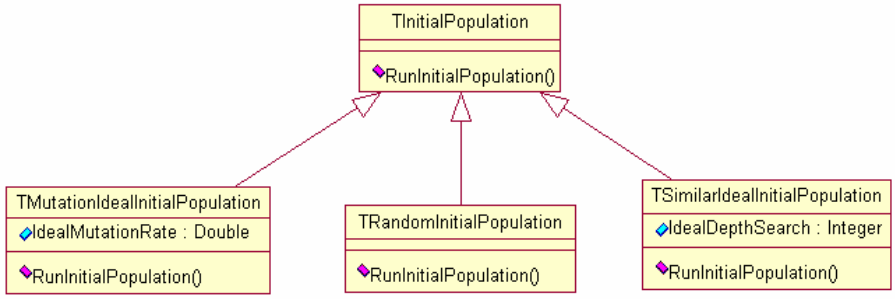

Figura 5 – Subclasses da classe *TInitialPopulation.*

Para a seleção dos indivíduos de uma população foram disponibilizadas três novas classes (*TRandomSelection*, *TBinaryTournamentSelection* e *TRouletteWhellSlection*) que estendem a classe *TSelection* do *framework*, como mostra a Figura 6. Todas as classes contêm o método *RunSelection()* que sobrepõe o da classe herdada, cada um contendo uma técnica diferente de seleção de indivíduos.

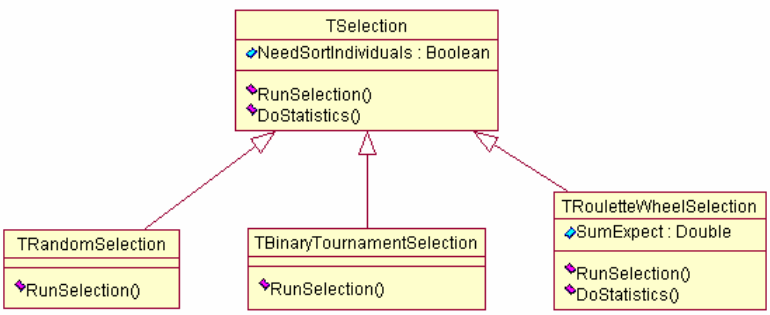

Figura 6 – Subclasses da classe *TSelection*.

Para realizar o cruzamento dos indivíduos foram disponibilizadas quator novas classes (*TOnePointCrossover*, *TTwoPointsCrossover*, *TAnyPointsCrossover* e *TArithmeticCrossover*) que estendem a classe *TMate* do *framework*, como mostra a Figura 7. Todas as classes contêm o método *RunMate()* que sobrepõe o da classe herdada, cada um contendo uma técnica diferente de cruzamento de indivíduos.

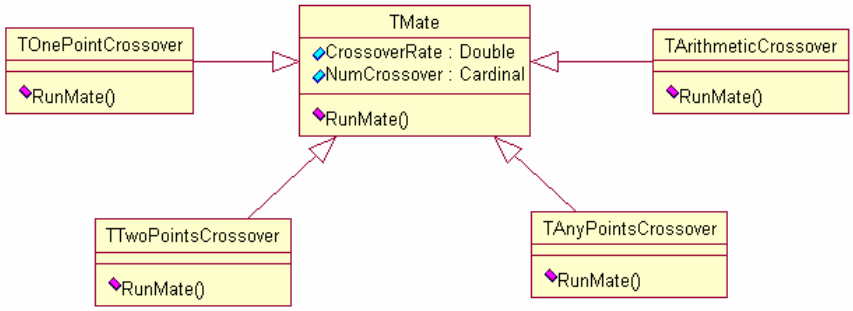

Figura 7 – Subclasses da classe *TMate*.

Para realizar a mutação nos indivíduos foi criada a classe *TRandomMutation*, herdada da classe *TMutation* do *framework*, como mostra a Figura 8. A classe contêm o método *RunMutation()* que sobrepõe o da classe herdada com uma técnica de mutação randômica.

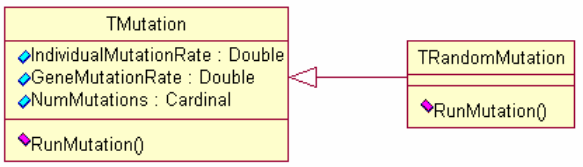

Figura 8 – Subclasses da classe *TMutation*.

4. CONCLUSÕES

O *framework* GeneticFrame mostrou-se eficaz no que diz respeito à extensibidade e reusabilidade do código no desenvolvimento de aplicações, tanto uniobijetivo, como multiobjetivo. Com isto promoveu a simplicidade e a facilidade de manutenção no código das aplicações desenvolvidas.

Pode-se observar através do exemplo mostrado, que a capacidade de extensão do *framework* é uniforme, permitindo um rápido desenvolvimento, uma vez que os principais métodos de controle e execução do algoritmo estão prontos. As classes *TSelection*, *TMate*, *TMutation e TInitialPopulation* mostram a versatilidade do modelo proposto, onde pode-se implementar os mais diversos tipos de operadores de seleção, cruzamento e mutação de indivíduos, incluindo a geração da população inicial.

Uma característica importante a ser observada no *framework* diz respeito ao baixo acoplamento das classes. A implementação de novos operadores, por exemplo, não implica na modificação ou extensão da classe *TCivilization*. Fica a cargo a aplicação que utiliza o *framework* selecionar e atribuir, em tempo de execução, quais são os operadores que serão utilizados pelo algoritmo genético. Esta é uma característica que pouco se vê nos *frameworks* citados na literatura.

5. REFERÊNCIAS BIBLIOGRÁFICAS

[1] GOLDBERG, D.; Genetic Algorithms in Search Optimization and Machine Learning, Addison Wesley 1989.

[2] SOARES, G.L.; VASCONCELOS, J.A., Algoritmos Genéticos: Estudo, Novas Técnicas e Aplicações. UFMG, Belo Horizonte, 1997.

[3] PACHECO, M. A., Notas de Aula em Computação Evolucionária. (www.ica.ele.puc-rio.br).

[4] GUIMARÃES, F.G.; RAMALHO, M.C., Implementação de Um Algoritmo Genético. UFMG, Belo Horizonte, 2001.

[5] SILVA, A.J., Implementação de um Algoritmo Genético utilizando o modelo de Ilhas. COPPE – UFRJ, Rio de Janeiro – RJ, 2005.

[6] JOHNSON, R. E.; FOOTE, B., Designing Reusable Classes. Journal of Object Oriented Programming – JOOP, 1(2):22-35, Junho/Julho 1988.

[7] COAD, P., Object-Oriented Patterns. Communications of the ACM, V. 35, nº 9, p 152-159, setembro 1992.

[8] FAYAD, M. C.; SCHMIDT, D. C.; JOHNSON, R. E., Building Application Frameworks – Object-Oriented foundations of frameworks design. Estados Unidos: Wiley, 1999.

[9] NEVES, T. A.; SOUZA, M.J.F.; MARTINS, A.X., Contrução de um protótipo de *framework* para otimização e seu uso na resolução do Problema de Roteamento de Veículos com Frota Heterogênea e Janelas de Tempo. UFOP, Outro Preto – MG.

[10] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J., Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos. Editora Bookman, Porto Alegre, 2000.

[11] Ziztler, E., Thiele, L., “Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach”. IEEE Transactions on Evolutionary Computation, 3(4): 257-271.