
ICA
Laboratório
Inteligência Computacional Aplicada
PUC
RIO
GACOM
Laboratório ICA - PUC-Rio
Departamento de
Engenharia Elétrica

Sumário

Introdução

O GACOM é uma biblioteca desenvolvida pelo ICA que tem como finalidade permitir o desenvolvimento de programas utilizando técnicas de computação evolucionária. Os objetivos desta biblioteca são:

- Reutilização: permitir que os módulos que compõem a biblioteca sejam facilmente intercambiáveis e reutilizáveis;
- Simplicidade: permitir, de maneira fácil e rápida, o desenvolvimento de novas aplicações;
- Extensibilidade: permitir que, se necessário, se acrescente novas funcionalidades a biblioteca sem a necessidade de se expor o funcionamento dos módulos já presentes.

Em sua atual versão, o GACOM conta com as seguintes funcionalidades: definição de diferentes tipos de representação de genes, segmentação do indivíduo para permitir cromossomos compostos por mais de um tipo de representação, formas de elitismo, seleção por roleta, normalização linear, operadores evolutivos (*crossover*, mutação e *crossover* especial), multiobjetivos com pareto, execução em pipeline para permitir inserção de blocos de código entre os passos da otimização, definição de diversos critérios de parada. Os demais operadores serão tratados no capítulo seguinte.

O componente possui suporte à restrições lineares (GENOCOP I) e não lineares (GENOCOP III), problemas com restrições de precedência (Problemas de planejamento), problemas co-evolucionário e multi-objetivos. Para esse último, recomenda-se a utilização do componente MULTICOM, também desenvolvido no ICA, o qual permite a resolução de problemas multi-objetivos através dos métodos de agregação de objetivos, distância ao alvo e minimização de energia.

Características

Genes

O componente permite a utilização de 5 tipos diferentes de gene conforme descritos na tabela 1.1. As classes que definem os genes se encontram no *namespace* `GACOM.Structures.GeneSpace`.

Tipo de Representação	**Classe (Structures.GeneSpace)**
Binária	BinaryGene
Matriz Binária	Binary2DGene
Inteiro	IntegerGene
Real	RealGene
Baseado em Ordem	OrderBasedGene
Real Genocop	GENOCOPRealGene

Tabela 1.1 – Tipos de representação de genes no GACOM

Segmentos

A implementação de cromossomos pode ser feita em diversos segmentos. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações, possibilitando que um cromossomo possa ter mais de um tipo de genes diferente. Os segmentos se encontram sob o *namespace* `GACOM.Structures.SegmentSpace` e podem ser dos seguintes tipos:

- Baseado em Ordem (`OrderBasedSegment`) – permite a representação de genes baseados em ordem. Para esse segmento é preciso inserir uma lista com o alfabeto que representa o problema.
- GENOCOP (`GenocopSegment`) – permite a representação de genes reais para a utilização do GENOCOP. Para esse segmento, passa-se o número de tentativas que o algoritmo deve executar até encontrar um indivíduo inicial que respeite as restrições, caso não seja possível, o usuário deve inserir uma semente inicial válida.

- Genérico (`Segment`) – permite a utilização de genes binários, inteiros ou reais.

Operadores Evolucionários

Operadores evolucionários consistem de métodos de cruzamento e mutação de indivíduos, com a finalidade de encontrar indivíduos mais aptos. Cada tipo de gene possui um conjunto de operadores relacionados a eles. No GAcom, deve-se atribuir a cada segmento, um conjunto de operadores que serão utilizados durante a evolução e a probabilidade de cada um desses operadores ser aplicados. A definição dessa probabilidade deve ser feita na criação do operador. Os operadores disponíveis são listados em seguida e se encontram sob o *namespace* `GACOM.EvolutionProcess.OperatorSpace`.

Crossover

Os operadores de cruzamento para segmentos binários e inteiros permitem a definição de um múltiplo do ponto de cruzamento, permitindo que o usuário defina que o cruzamento só ocorra em pontos múltiplos do número especificado. Isso pode ser feito através do construtor da classe. Os cruzamentos implementados para cada tipo de gene são:

- **Binário**
  - Cruzamento de um ponto (SinglePointCrossover) – realiza o crosover de um ponto (figura 1.1);

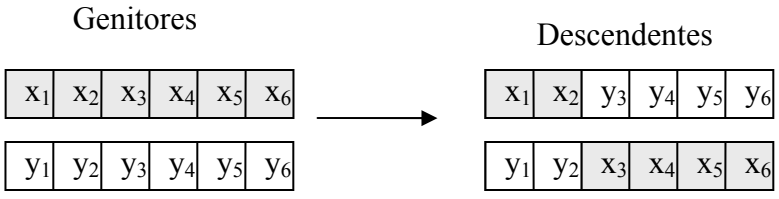


figura 1.1 – cruzamento de um ponto.

  - Cruzamento de dois pontos (TwoPointCrossover) – realiza o crosover de um ponto (figura 1.2);

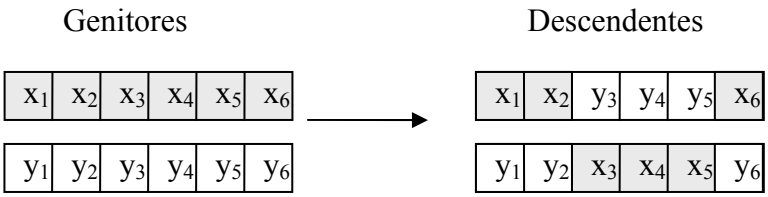


figura 1.2 – cruzamento de dois ponto.

- Cruzamento uniforme (UniformCrossover) – utiliza um padrão criado aleatoriamente para a geração dos filhos;

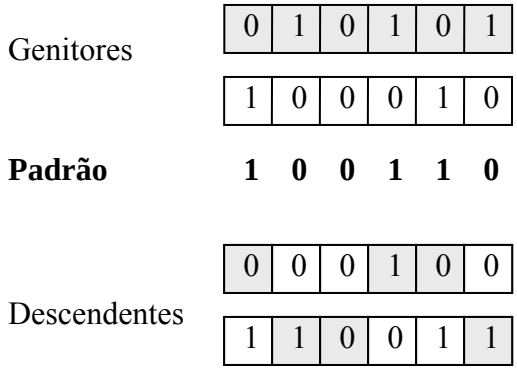


figura 1.3 – cruzamento uniforme.

- **Binário 2D**
  - Cruzamento de um ponto para matrizes 2D (SinglePoint2DCrossover);

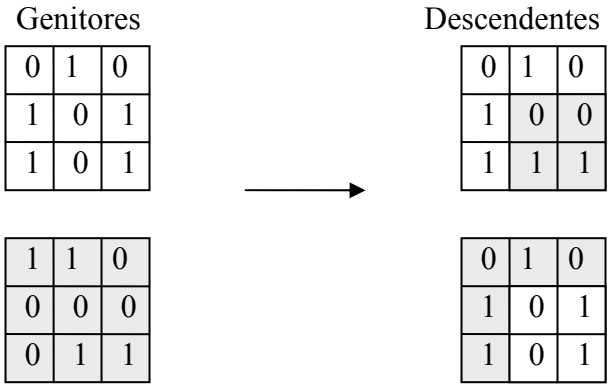


figura 1.4 – cruzamento de um ponto para matrizes 2D.

- **Inteiro**

Os operadores de cruzamento de genes inteiros seguem a mesma lógica que os binários, porém com representação inteira.

- Cruzamento de um ponto (SinglePointIntegerCrossover);
- Cruzamento de dois pontos (TwoPointIntegerCrossover);
- Cruzamento uniforme (IntegerUniformCrossover);

- **Real**
  - Aritmético (ArithmeticalCrossover) – realiza o crossover aritmético como demonstrado na equação a seguir:

$$F_1 = aP_1 + (1-a)P_2$$
$$F_2 = aP_2 + (1-a)P_1$$

    onde F_1 e F_2 são os filhos e P_1 e P_2 os pais;

- **GENOCOP**
  - GenocopSimpleCrossover – aplica o crossover aritmético para todos o genes a partir de um determinado índice sorteado aleatoriamente, garantindo que os descendentes respeitem as restrições lineares;
  - GenocopGeometricalCrossover;
  - GenocopSphereCrossover;
  - GenocopArithmeticCrossover – realiza o crossover aritmético em todos os genes das populações do GENOCOP garantindo que o resultado dessa operação respeite as restrições lineares;

- **Baseado em Ordem**
  - PMXCrossover (Partially-mapped Crossover) – gera descendentes ao escolher uma subseqüência da ordem de um dos pais e preservando a ordem do maior número possível de genes do outro pai;
  - OXCrossover (Order Crossover) – cria descendentes ao escolher uma subseqüência de um dos pais e mantendo a ordem relativa dos genes do outro pai;

- CXCrossover (Cycle Crossover) – gera descendentes de acordo com o cruzamento proposto por Oliver;

Mutação

- **Binário**
  - Mutação para 1 (AddBinaryMutation) – define um bit, escolhido aleatoriamente, do indivíduo como 1 (verdadeiro);
  - Mutação para 0 (SubBinaryMutation) – define um bit, escolhido aleatoriamente, do indivíduo como 0 (falso);
  - Inversão de bit (FlipBitMutation) – inverte o valor de um bit, escolhido aleatoriamente, do indivíduo (se verdadeiro torna falso e vice-versa);
- **Binário 2D**

  Os operadores de mutação de genes binários 2D seguem a mesma lógica que os binários simples.
  - Mutação para 1 (Add2DBinaryMutation);
  - Mutação para 0 (Sub2DBinaryMutation);
  - Inversão de bit (FlipBit2DMutation);
- **Inteiro**
  - Mutação uniforme (IntegerUniformMutation) – realiza a mutação baseada em uma máscara pseudo-aleatória, criada em função da probabilidade de aplicação do operador. Nessa operação os valores dos genes selecionados são substituídos por um valor aleatório entre os limites máximo e mínimo do gene. Pode-se também passar um delta pelo gerador do gene, indicando qual a variação máxima que o gene pode ter ao ser mutado;
- **Real**
  - Mutação uniforme (RealUniformMutation) – possui comportamento semelhante à mutação uniforme de inteiros;

- Mutação não uniforme (SimpleNonUniformMutation) – executa a mutação *creep*, conforme a equação a seguir, onde o valor do bit X na geração t+1 pode assumir dois valores distintos dependendo de um sorteio. Esse tipo de mutação faz com que o espaço de busca do operador nas gerações iniciais seja maior que nas gerações finais. O valor de b deve ser passado no construtor da classe, no campo *creep*; caso o usuário escolha por não defini-lo o valor padrão de 0.5 é utilizado;

$$X^{t+1} = \begin{cases} X^t + \Delta(t, \max - X^t) \\ X^t - \Delta(t, X^t - \min) \end{cases}$$

$$\Delta(t, s) = s.(1 - rand^{(1-t/T)^b})$$

Onde,

b é o grau de dependência com o número de geração;
t é a geração atual;
T é o número máximo de gerações;
rand é um número aleatório no intervalo [0,1].

- **Genocop**
  - GenocopUniformMutation – aplica a mutação uniforme garantindo que os descendentes respeitem as restrições lineares;
  - GenocopNonUniformMutation – aplica a mutação *creep* garantindo que os descendentes respeitem as restrições lineares;
  - GenocopBoundaryMutation – faz com que o gene da população de busca do GENOCOP assuma um dos valores limites (máximo ou mínimo);
  - GenocopGaussianMutation;
- **Baseado em Ordem**
  - PIMutation – inverte a ordem de um determinado intervalo de genes, os limites do intervalo são escolhidos aleatoriamente;
  - SwapMutation – sorteia dois genes do mesmo segmento e faz a troca dos seus valores;

- RotateLeftMutation – aplica a mutação rotacionando os genes para a esquerda;
- RotateRightMutation – aplica a mutação rotacionando os genes para a direita;

CrossoverSpecial

- GenocopHeuristicCrossover
- GenocopIIIArithmeticalCrossover – consiste de um cruzamento aritmético que é reproduzido até que um indivíduo que respeite todas restrições (lineares e não-lineares) seja alcançado. É atribuído à população de referência do GENOCOP.

Tipo de Avaliação

A avaliação de um indivíduo no GENOCOP pode ser feita através de um valor real (RealFitness ou DecimalFitness) ou inteiro (IntegerFitness). No caso da otimização de um problema de multi-objetivos, a avaliação pode ser apenas real (MultiRealFitness).

Inicialização da população

A inicialização da população pode ser feita através de três métodos localizador no *namespace* `GACOM.InitializePopulationSpace`:

- Inicializa cada indivíduo de forma aleatória (InitEachIndividual);
- Inicializa um indivíduo e o resto da população é criada como cópias desse indivíduo (InitOneIndividual). Bastante utilizado para problemas onde a criação de um indivíduo aleatoriamente é difícil (caso de problemas com restrições lineares e não-lineares;
- Inicializa cada indivíduo de forma a respeitar as restrições ineares do GENOCOP (InitIndividualGenocopI).

Saída de dados

As formas de saída dos dados pode ser feita através de arquivo, impressos na tela ou então retornando o melhor individuo, a média e o valor de offline de cada geração para que o usuário possa manipulá-las em tempo real. As classes disponíveis para saída de dados são: FileOutput, CoevolutionaryGAFileOutput (para problemas coevolucionários), MultiFileOutput (para problemas multiobjetivos), SimpleGAConsoleOutput, CoevolutionaryGAConsoleOutput (para problemas coevolucionários) e OnlineOutput.

Seleção de genitores e operadores

A forma disponível para a seleção tanto dos genitores quanto dos operadores, é a seleção por roleta (RouletteGenitorSelection e RouletteOperatorSelection, respectivamente), através dela, os indivíduos mais aptos possuem maior probabilidade de serem selecionados para sofrerem operações evolutivas.

Operadores

O GACOM disponibiliza uma série de operadores e variáveis, trataremos agora de cada um deles e sua aplicabilidade.

Geradores de Números Aleatórios

Existem inúmeros tipos de geradores de números aleatórios, de forma a auxiliar o usuário, o GACOM disponibiliza 6 tipos de geradores de números pseudo-aleatórios distintos e 3 quase-aleatórios, além da possibilidade de utilização do gerador padrão do c#, são eles:

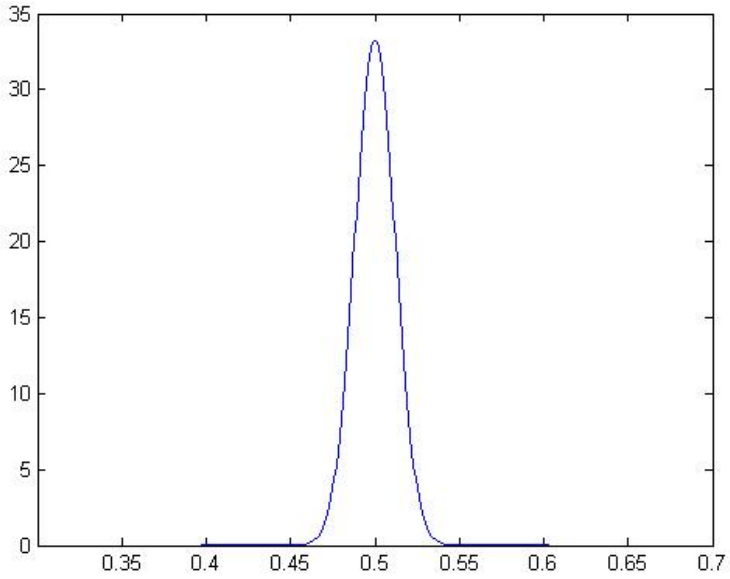
- LCG4RandGen: implementa um gerador usando quatro geradores lineares congruentes (linear congruential generator – LCG) desenvolvido por Pierre L'Ecuyer. Possui um período de 2^{121} valores distintos;
- LGMRandGen: implementa o algoritmo de Lewis-Goodman-Miller(1969), o qual é um simples gerador linear congruente. Possui um período de (2^{31})-2 valores distintos;
- MRG32k3aRandGen: implementa o gerador recursivo multimplo (Multiple Recursive Generators) desenvolvido por Pierre L'Ecuyer. Possui um período de 2^{191} valores distintos;
- MRG32k5aRandGen: implementa o gerador recursivo multimplo (Multiple Recursive Generators) desenvolvido por Pierre L'Ecuyer. Possui um período de 2^{319} valores distintos;
- MRG63k3aRandGen: implementa o gerador recursivo multimplo (Multiple Recursive Generators) desenvolvido por Pierre L'Ecuyer. Possui um período de 2^{377} valores distintos;
- VS2GAComRandGen: Utiliza o gerador aleatório da framework.net.
- QMCHaltonRandGen: implementa o algoritmo de Halton Quasi Monte Carlo;
- QMCSobol64RandGen: implementa o algoritmo de Sobol Quasi Monte Carlo;
- QMCSobolRandGen: implementa o algoritmo de Sobol Quasi Monte Carlo;

Funções de Distribuição

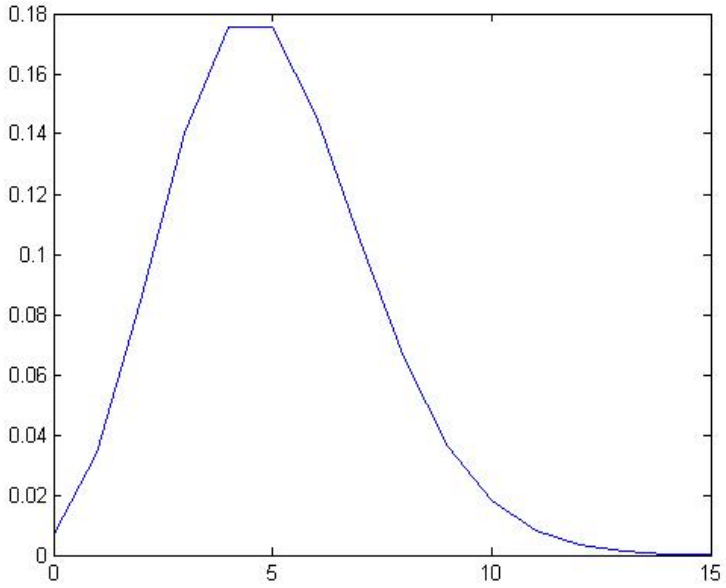
Estes geradores de números aleatórios seguem uma distribuição que também deve ser definida pelo usuário. Disponibiliza 5 tipos de distribuições:

- Normal (*NormalSampler*) – gera números aleatórios baseados na distribuição normal. Pode-se passar o valor da média e da variância da distribuição, caso contrário utiliza valores padrões para média (μ=0.5) e variância (σ=0.012). A distribuição se apresenta conforme a equação e figura abaixo;

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$



- Poisson (*PoissonSampler*) – gera números baseado na distribuição de Poisson. Deve-se passar como parâmetro da função o valor do lambda, caso contrário λ=0 é utilizado. A figura abaixo ilustra a distribuição;



- uniforme (*UniformSampler*) – A distribuição uniforme é a distribuição de probabilidades contínua mais simples de conceituar: a probabilidade de se gerar qualquer ponto em um intervalo contido no espaço amostral é proporcional ao tamanho do intervalo. Seja [a,b] o espaço amostral, então temos que:

$$f(x) = \frac{1}{b-a}, \quad \text{se } a \le x \le b$$

$$f(x) = 0, \quad \text{se } x < a \text{ ou } x > b$$

Pode-se passar os valores do limite superior e inferior da distribuição através do seu construtor. Caso o usuário não defina os limites, a distribuição assume o intervalo [0, 1];

- trapezoidal (*TrapezoidalSampler*) – gera números baseados na distribuição trapezoidal. O usuário deve indicar os quatro vértices do trapézio (mínimo, dois vértices médios e máximo). Caso não indique os valores desejados, os valores default são: min=0, med1=0.25, med2=0.75 e Max = 1;

- triangular (*TriangularSampler*) – gera números baseados na distribuição triangular. Os pontos do triângulo podem ser definidos pelo usuário, caso contrário os

valores de 0, 0.5 e 1 são atribuídos aos vértices mínimo, médio e máximo, respectivamente;

Operadores de Elitismo

O operador de elitismo implementado no GACOM é o SteadyState. Para esse operador, deve-se indicar o GAP correspondente ao método. O GAP do Steady State indica quantos indivíduos novos serão gerados a cada geração. Caso esse valor seja menor que 1, corresponderá à quantidade percentual de indivíduos novos que serão criados.

Instalando o GACOM

A instalação do GACOM é simples e consiste basicamente em se copiar o arquivo GACOM.dll dentro de alguma pasta (preferencialmente, a pasta \WINDOWS\SYSTEM32). Alguns passos, no entanto, se fazem necessários após a cópia do arquivo:

1. Após criar (ou abrir) o seu projeto dentro do .NET, usando o *Solution Explorer* clique com o botão direito na pasta *References* e selecione *"Add Reference ..."*;
2. Selecione a aba .NET e clique no botão "*Browse...*". Selecione a DLL do GACOM na pasta em que a mesma foi salva.
3. Clique OK. A biblioteca está pronta para ser utilizada pela sua aplicação.

Interface com o MATLAB

O GACOM permite a avaliação dos indivíduos chamando funções no MATLAB. Para isso deve-se utilizar uma biblioteca que permita a interface com o MATLAB. Um exemplo desse tipo de implementação pode ser visto no tutorial da representação baseada em ordem, onde a biblioteca EngMATLib é utilizada. Ela disponibiliza funções para simplificar o acesso às variáveis do MATLAB, as mais usuais são:

Evaluate() – Avalia uma expressão;

EvaluateAsString() – Avalia uma expressão;

GetMatrix() – Retorna uma variável do MATLAB;

SetMatrix() – Atribui um valor a uma variável do MATLAB;

Matrix() – Cria uma matriz no MATAB.

Criando avaliações distribuídas

O GACOM permite que a avaliação dos indivíduos seja feita de forma distribuída, para isso é preciso setar o flag isDistributed da avaliação como verdadeiro. O código abaixo ilustra essa configuração.

```
////////////////////////////////////////////////////
// Avaliação
////////////////////////////////////////////////////

Evaluation eval = new Evaluation(dec);
eval.isDistributed = true;
```

Após a configuração do flag, deve-se agora, na avaliação, sobrescrever o método evaluate de forma distinta, segundo o código abaixo.

```
public override Fitness evaluate(IGene[] objects, ref IIndividual[]
      individuals, IIndividual otherPopIndividual, Fitness[]
      individualsFitness, int generation)
{
```

Assim, durante a avaliação o GACOM irá passar toda a população para a função de avaliação, para que ela seja avaliada todas de uma vez pelo método sobrescrito, possibilitando que o usuário distribua a avaliação como bem entender.

Algoritmos Evolutivos com Inspiração Quântica

Proposto por André Cruz (2007), trata de um algoritmo genético com duas populações, uma que uma população de indivíduos que representa a superposição dos possíveis estados, em que os genes dos indivíduos são considerados funções de densidade de probabilidade uniforme, com centro e largura definindo o indivíduo (população quântica); e outra com indivíduos observados a partir dessas funções (população clássica). Como na física quântica, os indivíduos da população quântica não possuem aptidão, uma vez que não podem ser medidos; essa tarefa é feita pelos indivíduos clássicos gerados a partir da população quântica.

O algoritmo busca um tratamento para problemas numéricos a fim de diminuir o tempo de convergência e, mesmo assim, encontrar o ótimo. A listagem completa do algoritmo evolutivo com inspiração quântica se encontra abaixo, onde Q(t) é a população quântica e E(t), a clássica.

```
iniciar
01. t ← 1
02. Gerar população quântica Q(t) com N indivíduos com G genes
03. enquanto (t <= T)
04.   E(t) ← gerar indivíduos clássicos observando indivíduos quânticos
05.   se (t=1) então
06.     C(t) ← E(t)
07.   senão
08.     E(t) ← recombinação entre E(t) e C(t)
09.     avaliar E(t)
10.     C(t) ← K melhores indivíduos de [E(t) ∪ C(t)]
11.   fim se
12. Q(t + 1) ← Atualiza Q(t) usando os N melhores indivíduos de C(t)
13. t ← t + 1
14. fim enquanto
fim
```

A seção Tutoriais possui um exemplo de utilização do algoritmo genético quântico (GA quântico).

Tutoriais

Com o intuito de auxiliar os usuários que estão tendo seu primeiro contato com o GAcom, esse tutorial apresenta a estrutura básica para a utilização do componente, assim como alguns casos de uso para genes binários, reais, baseados em ordem e genocop.

Para a utilização do GAcom a seguinte estrutura básica deve ser seguida para todos os casos de implementação:

1. Definir os pacotes que deseja utilizar;
2. Definir a classe que irá herdar a classe GA do GACOM;
3. Sobrescrever o método `Init()` desta classe, que, por sua vez, deverá definir:
   (a) Os geradores de números aleatórios que serão utilizados pelo algoritmo genético;
   (b) Os amostradores, com suas respectivas distribuições de probabilidade;
   (c) O operador de elitismo que o algoritmo genético irá usar;
   (d) A seleção dos operadores genéticos;
   (e) Os operadores genéticos;
   (f) A classe de reprodução;
   (g) O operador de seleção de genitores;
   (h) A classe de evolução;
   (i) O decodificador;
   (j) A classe de avaliação;
   (k) O modo de inicialização dos indivíduos;
   (l) O modelo de trace;
   (m) A população e todas as suas definições, o que inclui:
      i. Se necessário, o operador de normalização linear;
      ii. Os indivíduos que formam a população;
   (n) As condições de parada;
   (o) O método de exibição de resultados;
4. Definir a classe que herda de *FitnessFuction*.

Representação Binária

O primeiro exemplo do tutorial é uma aplicação simples utilizando a representação binária. O objetivo deste exemplo é maximizar a função F6, definida pela equação (1) e representada graficamente pela figura 1.

$$F6(x,y) = 0.5 - \frac{\left(\sin\sqrt{x^2+y^2}\right)^2 - 0.5}{1.0 + 0.001\left(x^2+y^2\right)^2} \qquad (1)$$

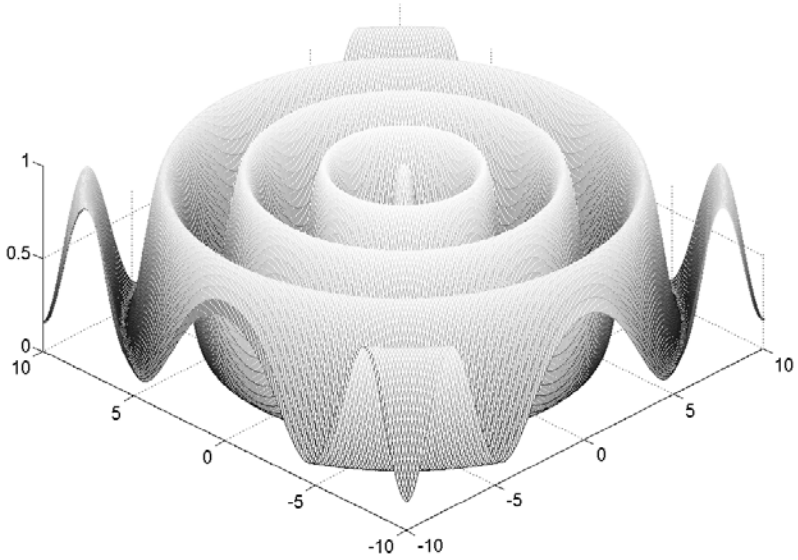


Figura 1. Representação gráfica da função F6.

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto mais comum:

```
using System;
using System.Collections;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.EvolutionProcess.OperatorSpace.BinaryOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
```

```
using GACOM.EvaluationProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.OutputSpace;
using GACOM.StopConditionSpace;
using GACOM.EvolutionProcess.RateSpace;
```

Em seguida, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class GAF6 : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public GAF6(string name)
    :
    base(name)
{ }
```

Sobrescreve-se o método `Init()`, responsável pelas configurações do algoritmo genético:

```
public override void Init()
{
```

Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;
p_random = new IRandom[2];
p_random[0] = new LGMRandGen();
p_random[1] = new MRG32k3aRandGen();
```

Neste caso foram criados dois geradores de número aleatório, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de número aleatório criados anteriormente. Para tanto, deve-se fazer como exposto abaixo:

```
samplers = new ISampler[2];
samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Pode-se também definir a técnica de seleção, nesse caso foi usado o *steady-state*. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez,

recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Para isso, pode-se utilizar o comando abaixo:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
   initial_steadystate_, final_steadystate_), false);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso, seleção por roleta) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, são usados os operadores de mutação *flip-bit* (o qual inverte o bit correspondente ao gene, ou seja se o bit for 1 vira 0 e vice-versa) *add-bit* (atribui 1 ao gene) e *sub-bit* (atribui 0 ao gene) e os cruzamentos uniforme, de um e de dois pontos. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
   FinalCrossover);
LinearRate mutationRate = new LinearRate(this, InitialMutation,
   FinalMutation);

Operator CroOp1 = new UniformCrossover(crossoverRate);
CroOp1.Sampler = samplers[1];

Operator CroOp2 = new TwoPointCrossover(crossoverRate);
CroOp2.Sampler = samplers[1];

Operator CroOp3 = new SinglePointCrossover(crossoverRate);
CroOp3.Sampler = samplers[1];

Operator MutOp1 = new FlipBitMutation(mutationRate);
MutOp1.Sampler = samplers[1];

Operator MutOp2 = new AddBinaryMutation(mutationRate);
MutOp2.Sampler = samplers[1];

Operator MutOp3 = new SubBinaryMutation(mutationRate);
MutOp3.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção. O GAcom permite ao usuário a definição de um peso para cada operador. Esse peso é passado como parâmetro ao adicionar o operador. Alem disso o componente possibilita a utilização de segmentos distintos. Segmentos são utilizados pelo GAcom para separar partes do cromossomo com diferentes representações. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento e com o mesmo peso:

```
opsel1.addOperator(MutOp1, 1, 0);
opsel1.addOperator(MutOp2, 1, 0);
opsel1.addOperator(MutOp3, 1, 0);
opsel1.addOperator(CroOp1, 1, 0);
```

```
opsel1.addOperator(CroOp2, 1, 0);
opsel1.addOperator(CroOp3, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
TesteEvaluation evaluationFunc = new TesteEvaluation();
Decoder dec = new Decoder(evaluationFunc);
Evaluation eval = new Evaluation(dec);
InitEachIndividual indInit = new InitEachIndividual();
Population pop = new Population(PopulationSize, "pop", eval,
                                                evol1, this, indInit);
ITrace trace_model = new Trace();
pop.TraceModel = trace_model;
pop.Maximization = true;
```

A primeira linha instancia a classe de reprodução indicando os operadores genéticos escolhidos e o *steady-state*. A linha seguinte define o modelo de seleção dos genitores (no caso, por roleta), e passa o amostrador para o mesmo. Em seguida, instancia-se a classe que controla a evolução, passando como parâmetros a técnica de seleção e a reprodução. Na linha seguinte, tem-se a definição do decodificador que será utilizado pelo algoritmo genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. A população recebe como parâmetros a quantidade de indivíduos que a constituem (tamanho da população), uma string utilizada para identificá-la, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. Por último, instancia-se o modelo de trace, que guarda informações relativas à população, e define-se o problema como de maximização.

Após a criação da população, deve-se ainda adicionar o método de normalização linear e adicioná-la a um bloco, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                        SimpleLinearNormalization(1, PopulationSize);
pop.addBlock(2, norm);
```

Finalmente, deve-se preencher a população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
private void fillPopulations(ref Population population)
{
    Individual ind = null;
    Segment seg1 = null;
    Gene gene = null;
```

```
        ind = new Individual(1);
        seg1 = new Segment(TamCromo);

        for (int j = 0; j < seg1.TotalGenes; j++)
        {
              gene = new BinaryGene();
              seg1.AddGene(gene);
        }

        ind.AddSegment(seg1);

        ind.Fitness = new RealFitness();
        ind.OriginalFitness = new RealFitness();

        population.addIndividual(ind);
        for (int j = 0; j < PopulationSize - 1; j++)
        {
              population.addIndividual(Individual)ind.Clone());
        }
  }
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado o número de segmentos que este possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Neste caso, como todo o cromossomo é binário, o indivíduo possui apenas um segmento. Para a criação deste, é passado o números de genes que o constituirão. Após criar o segmento, adiciona-se os genes ao mesmo. Feito isso, adiciona-se o segmento ao indivíduo. As duas linhas seguintes servem para indicar ao GACOM que tanto a avaliação (*OriginalFitness*) quanto a aptidão (*Fitness*) são valores reais.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método Clone. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Agora, deve-se adicionar a população recém-criada ao algoritmo genético. Isto pode ser eito através do seguinte comando:

```
addPopulation(pop);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
SimpleGAConsoleOutput cvout = new SimpleGAConsoleOutput(this);
```

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TesteEvaluation : FitnessFunction
{
```

O método *evaluate* da classe FitnessFuction deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objs,
    ref IIndividual individual, int generation)
{
```

O código completo desta classe é mostrado abaixo:

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;

public class TesteEvaluation : FitnessFunction
{
  public TesteEvaluation()
  {}

  private const double PI = 3.141592654;

  public override Fitness evaluate(IGene[] objs,
    ref IIndividual individual, int generation)
  {
    if (objs.Length == 44)
    {
      // Faz a decodificação de binário para real
      double x = 0;
      double y = 0;
      for (int i = 0; i < objs.Length / 2; i++)
      {
        x += (Convert.ToDouble(objs[i].Value)) * Math.Pow(2, i);
        y += (Convert.ToDouble(objs[i + objs.Length/2].Value)) *
Math.Pow(2, i);
      }
      x = (x * (100 + 100) / (Math.Pow(2, 22) - 1)) - 100;
      y = (y * (100 + 100) / (Math.Pow(2, 22) - 1)) - 100;

      individual.OriginalFitness = new RealFitness(f6(x, y));
      individual.Fitness = new
          RealFitness((double)individual.OriginalFitness.Value);

        return new
RealFitness((double)individual.OriginalFitness.Value);
      }
    return null;
  }

  public double f6(double x, double y)
  {
```

```
        double dividendo;
        double divisor;
        double r1;

        dividendo = Math.Pow(Math.Sin((Math.Sqrt(x * x + y * y) * PI) /
180), 2) - 0.5;
        divisor = Math.Pow(1 + 0.001 * (x * x + y * y), 2);

        r1 = 0.5 - (dividendo / divisor);

        return r1;
    }
}
```

Por se tratar de uma representação binária de um problema real, antes de fazer sua avaliação (método `f6()`) é preciso decodificar o indivíduo binário em valores reais. Então deve-se definir o *fitness* do indivíduo como o resultado dessa avaliação.

Por fim, precisamos definir o programa principal que irá chamar o GAcom. A seguir temos um exemplo de programa que chama o GACOM:

```
using System;

class Program
{
    static void Main(string[] args)
    {
          GAF6 ga = new GAF6("F6");

          ga.Generations = 80;
          ga.PopulationSize = 40;
          ga.InitialCrossover = 0.8;
          ga.FinalCrossover = 0.65;
          ga.InitialMutation = 0.08;
          ga.FinalMutation = 0.8;
          ga.InitialSteadyState = 0.4;
          ga.FinalSteadyState = 0.4;
          ga.Experiments = 20;

          ga.exec();
    }
}
```

Representação Real

Agora, utilizaremos a mesma função do exemplo anterior, porém com representação real. A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto utilizado, enfatizando que agora utilizaremos genes reais:

```
using System;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.EvolutionProcess.OperatorSpace.RealOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.OutputSpace;
using GACOM.StopConditionSpace;
using GACOM.EvolutionProcess.RateSpace;
```

A classe que herdará a classe GA presente na biblioteca do GACOM, seguirá o mesmo padrão do tutorial anterior, contendo suas principais modificações nos operadores utilizados e na criação da população inicial. Ilustraremos apenas as modificações, os códigos completos podem ser vistos nos arquivos contidos em "`F6_Real.zip`".

Os operadores utilizados agora são: mutação simples não-uniforme, mutação uniforme real e cruzamento aritmético. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
   FinalCrossover);
LinearRate mutationRate = new LinearRate(this, InitialMutation,
   FinalMutation);

Operator CroOp1 = new ArithmeticalCrossover(crossoverRate);
CroOp1.Sampler = samplers[1];

Operator MutOp1 = new RealUniformMutation(mutationRate);
MutOp1.Sampler = samplers[1];

Operator MutOp2 = new SimpleNonUniformMutation(mutationRate, 0.7,
this);
MutOp2.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento:

```
opsel1.addOperator(MutOp1, 1, 0);
opsel1.addOperator(MutOp2, 1, 0);
opsel1.addOperator(CroOp1, 1, 0);
```

A próxima modificação significativa em relação ao exemplo anterior está no preenchimento da população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
private void fillPopulations(ref Population population)
{
  Individual ind = null;
  Segment seg1 = null;
  Gene gene = null;

  int min_value_clock = -100;
  int max_value_clock = 100;

  ind = new Individual(1);
  seg1 = new Segment(TamCromo);

  for (int j = 0; j < TamCromo; j++)
  {
    gene = new RealGene(min_value_clock, max_value_clock);
    seg1.AddGene(gene);

  }

  ind.AddSegment(seg1);

  ind.Fitness = new RealFitness();
  ind.OriginalFitness = new RealFitness();

  population.addIndividual(ind);

  for (int j = 0; j < PopulationSize - 1; j++)
  {
    population.addIndividual((IIndividual)ind.Clone());
  }
}
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Para criar um indivíduo é preciso passar como parâmetro o número de segmentos que este indivíduo possuirá. Neste caso, como todo o cromossomo é real, o indivíduo possui apenas um segmento. Para instanciar um segmento é preciso passar a quantidade de genes que este possuirá. Após criar o segmento, adiciona-se os genes ao mesmo, cuidando para indicar os limites mínimos e máximos que o gene pode ter (neste caso, o intervalo [-100,100]). Feito isso, adiciona-se o segmento ao indivíduo. As duas linhas seguintes servem para indicar ao GACOM que tanto a avaliação (*OriginalFitness*) quanto a aptidão (*Fitness*) são valores reais.

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TesteEvaluation : FitnessFunction
{
```

O método *evaluate* da classe FitnessFuction deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objs,
    ref IIndividual individual, int generation)
{
```

A código completo desta classe é mostrado abaixo. Nota-se que, por tratar de uma representação real, não é preciso converter os dados.

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;

public class TesteEvaluation : FitnessFunction
{
  public TesteEvaluation()
  { }

  private const double PI = 3.141592654;

  public override Fitness evaluate(IGene[] objs,
      ref IIndividual individual, int generation)
  {
    if (objs.Length == 2)
    {
      IGene gene1 = (IGene)objs[0];
      IGene gene2 = (IGene)objs[1];

      individual.OriginalFitness = new
          RealFitness(f6((double)gene1.Value, (double)gene2.Value));

      individual.Fitness = new
          RealFitness((double)individual.OriginalFitness.Value);

      return new RealFitness((double)individual.OriginalFitness.Value);
    }
    return null;
  }

  public double f6(double x, double y)
  {
    double dividendo;
    double divisor;
    double r1;
```

```
    dividendo = Math.Pow(Math.Sin((Math.Sqrt(x * x + y * y) * PI) /
180), 2) - 0.5;
    divisor = Math.Pow(1 + 0.001 * (x * x + y * y), 2);

    r1 = 0.5 - (dividendo / divisor);

    return r1;
  }
}
```

GENOCOP – Restrições Lineares

Nesse exemplo, faremos a minimização da equação (2), cujo ponto ótimo se encontra em -213:

$$f(x, y) = -10.5x_1 - 7.5x_2 - 3.5x_3 - 2.5x_4 - 1.5x_5 - 10y - 0.5\sum_{i=1}^{5} x_i^2 \quad (2)$$

onde,

$$6x_1 + 3x_2 + 3x_3 + 2x_4 + x_5 \leq 6.5, \qquad 10x_1 + 10x_3 + y \leq 20,$$
$$0 \leq x_i \leq 1, \qquad 0 \leq y.$$

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. Para a utilização do GENOCOP alguns pacotes devem ser adicionados, estes se encontram destacados no código abaixo, o qual mostra o conjunto utilizado nesse exemplo:

```
using System;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.EvolutionProcess.OperatorSpace.GenocopOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.OutputSpace;
using GACOM.StopConditionSpace;
using GACOM.EvolutionProcess.RateSpace;
using GACOM.ConstraintsSpace;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class TesteGA : GA
{
```

Dentro desta classe, deve-se sobrescrever o método `Init()`. A definição do método é dada abaixo:

```
public override void Init()

{
```

Agora, inicia-se a construção do corpo do método construtor. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;
p_random = new IRandom[2];
p_random[0] = new MRG63k3aRandGen();
p_random[1] = new MRG63k3aRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Pode-se também definir o *steady-state*, caso seja necessária a utilização do mesmo. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Para isso, pode-se utilizar o comando abaixo:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
                   initial_steadystate_, final_steadystate_), false);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso, seleção or roleta) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, são usados os operadores de mutação uniforme, não uniforme, de fronteira e crossover aritmético, simples e heurístico do GENOCOP. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
FinalCrossover);

// Operadores do segmento 1
Operator MutOp11 = new GenocopUniformMutation(mutationRate);
MutOp11.Sampler = samplers[1];

Operator MutOp12 = new GenocopNonUniformMutation(mutationRate, this);
MutOp12.Sampler = samplers[1];

Operator MutOp13 = new GenocopBoundaryMutation(mutationRate);
MutOp13.Sampler = samplers[1];

Operator CroOp11 = new GenocopArithmeticalCrossover(crossoverRate);
CroOp11.Sampler = samplers[1];

Operator CroOp12 = new GenocopSimpleCrossover(crossoverRate);
CroOp12.Sampler = samplers[1];

Operator CroOp15 = new GenocopHeuristicCrossover(crossoverRate, 100);
CroOp15.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento:

```
opsel1.addOperator(MutOp11, 1, 0);
opsel1.addOperator(MutOp12, 1, 0);
opsel1.addOperator(MutOp13, 1, 0);
opsel1.addOperator(CroOp11, 1, 0);
opsel1.addOperator(CroOp12, 1, 0);
opsel1.addOperator(CroOp15, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
evaluationFunc = new TesteEvaluation();
Decoder dec = new Decoder(evaluationFunc);
Evaluation eval = new Evaluation(dec);
InitOneIndividual indInit = new InitOneIndividual();
Population pop1 = new Population(pop_size_, "pop1", eval, evol1, this,
indInit);
pop1.Maximization = false;
ITrace trace_model = new Trace();
pop1.TraceModel = trace_model;
```

A primeira linha instancia a classe de reprodução, indicando os operadores genéticos escolhidos e o *steady-state*. A linha seguinte define o modelo de seleção dos genitores (no caso, por roleta), e passa o amostrador para o mesmo. Em seguida, instancia-se a classe que controla a evolução, passando como parâmetros a técnica de seleção e a reprodução. Na linha seguinte, tem-se a definição do decodificador que será utilizado pelo algoritmo

genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. A população recebe como parâmetros o tamanho da população, uma string utilizada para identificá-la, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. A variável *Maximization* é atribuída *false*, pois queremos minimizar a função. Por último, instancia-se o modelo de trace, que guarda informações relativas à população.

Após a criação da população, deve-se ainda adicionar o método de normalização linear, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                        SimpleLinearNormalization(1, PopulationSize);
pop1.addBlock(2, norm);
```

Finalmente, deve-se preencher a população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
fillPopulations(ref pop1);

private void fillPopulations(ref Population population)
{
  Individual ind = null;
  GenocopSegment seg1 = null;
  RealGene gene = null;

  double min_value_clock = 0.0;
  double max_value_clock = 1.0;

  IneqLinearConstraints ilc1 = new
                         IneqLinearConstraints("linear.txt", 6);

  ind = new Individual(1);
  seg1 = new GenocopSegment(6, 10000, ilc1);

  for (int j = 0; j < 6 - 1; j++)
  {
    gene = new RealGene(0, min_value_clock, max_value_clock);
    seg1.AddGene(gene);
  }

  gene = new RealGene(0, 0, 50);
  seg1.AddGene(gene);

  ind.AddSegment(seg1);

  ind.Fitness = new RealFitness();
  ind.OriginalFitness = new RealFitness();
  ind.SpecieName = "clock";
```

```
    population.addIndividual(ind);

    for (int j = 0; j < PopulationSize - 1; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado como parâmetro a quantidade de segmentos que esse possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Neste caso, como todo o cromossomo é real, o indivíduo possui apenas um segmento. Na criação do segmento, deve-se indicar o número de genes, o número máximo de tentativas para gerar um indivíduo válido e as restrições lineares. Após criar o segmento, adiciona-se os genes ao mesmo, cuidando para indicar os limites mínimos e máximos que o gene pode ter (neste caso, o intervalo [0,1]). As restrições lineares são passadas através de um arquivo, neste caso o "linear.txt". O arquivo deve ser criado de forma que cada linha corresponda a uma inequação do tipo "x_1 x_2 x_3 x_4 x_n", onde $x_1 + x_2 + x_3 + x_4 + x_n \geq 0$.

Feito isso, adiciona-se o segmento ao indivíduo. As duas linhas seguintes servem para indicar ao GACOM que tanto a avaliação (*OriginalFitness*) quanto a aptidão (*Fitness*) são valores reais.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método *Clone*. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Agora, deve-se adicionar a população recém-criada ao algoritmo genético. Isto pode ser feito através do seguinte comando:

```
addPopulation(pop1);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
Experiments = 0;
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
SimpleGAConsoleOutput cvout = new SimpleGAConsoleOutput(this);
cvout.ShowIntraOutput = true;
Output = cvout;
```

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TesteEvaluation : FitnessFunction
{
```

O método `evaluate()` da classe *FitnessFuction* deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objs,
                        ref IIndividual individual, int generation)
{
```

O código completo desta classe é mostrado abaixo:

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;
using GACOM.Structures.SegmentSpace;

public class TesteEvaluation : FitnessFunction
{
  public TesteEvaluation()
  { }

  private const double PI = 3.141592654;

  public override Fitness evaluate(IGene[] objs,
                ref IIndividual individual, int generation)

  {

    if (objs.Length == 6)
    {
      IGene gene1 = (IGene)objs[0];
      IGene gene2 = (IGene)objs[1];
      IGene gene3 = (IGene)objs[2];
      IGene gene4 = (IGene)objs[3];
      IGene gene5 = (IGene)objs[4];
      IGene gene6 = (IGene)objs[5];

      individual.OriginalFitness = new
RealFitness(tc1((double)gene1.Value, (double)gene2.Value,
(double)gene3.Value, (double)gene4.Value, (double)gene5.Value,
(double)gene6.Value));

      for (int i = 0; i < individual.TotalSegments; i++)
        ((GenocopSegment)individual.getSegmentAt(i)).FitnessValue =
individual.OriginalFitness.Value;

      individual.Fitness = new
             RealFitness((double)individual.OriginalFitness.Value);

      return new RealFitness((double)individual.OriginalFitness.Value);
    }
```

```
    return null;
  }

  public double tc1(double x1, double x2, double x3, double x4, double
x5, double y)
  {
    double d1, d2;

    d1 = -10.5*x1 - 7.5*x2 - 3.5*x3 - 2.5*x4 - 1.5*x5 - 10*y;
    d2 = 0.5*(Math.Pow(x1, 2) + Math.Pow(x2, 2) + Math.Pow(x3, 2) +
Math.Pow(x4, 2) + Math.Pow(x5, 2));

    return (d1 - d2);
  }
}
```

Para ilustrar, temos um programa que utiliza o GACOM com alguns parâmetros configurados:

```
using System;
using GACOM;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.IndividualSpace;

class TestingGA
{
  static void Main(string[] args)
  {
    TesteGA ga = new TesteGA("TesteGA");

    ga.Generations = 1000;
    ga.PopulationSize = 70;
    ga.InitialCrossover = 0.9;
    ga.FinalCrossover = 0.75;
    ga.InitialMutation = 0.3;
    ga.FinalMutation = 0.5;
    ga.InitialSteadyState = 0.3;
    ga.FinalSteadyState = 0.8;
    ga.Experiments = 1;
    ga.exec();

    Console.ReadKey();
  }
}
```

GENOCOP III – Restrições Não-Lineares

Nesse exemplo, é feita a minimização da seguinte função não linear:

$$G3(x) = (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 +$$

$$10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7$$

Onde,

$$127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geq 0,$$
$$282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 \geq 0,$$
$$196 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 \geq 0,$$
$$-4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 \geq 0$$
$$-10.0 \leq x_i \leq 10.0, i = 1,...,7$$

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto utilizado:

```
using System;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.EvolutionProcess.OperatorSpace;
using GACOM.EvolutionProcess.OperatorSpace.GenocopOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSpace.GenocopIIIOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess;
using GACOM.EvolutionProcess.CoevolutionaryIndividualSelection;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.OutputSpace;
using GACOM.StopConditionSpace;
using GACOM.EvolutionProcess.RateSpace;
using GACOM.ConstraintsSpace;
using System.Collections;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class TesteGA : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public TesteGA(string name, IStopCondition stop_condition)
  :
base(name, stop_condition) { }
```

Agora, inicia-se a construção do corpo do método construtor. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;

p_random = new IRandom[2];
p_random[0] = new LGMRandGen();
p_random[1] = new LGMRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Pode-se também definir o *steady-state*, caso seja necessária a utilização do mesmo. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Para isso, pode-se utilizar o comando abaixo:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
                   initial_steadystate_, final_steadystate_), true);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso, seleção or roleta) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, são usados diversos operadores, entre eles os operadores de mutação uniforme, não uniforme, de fronteira, gaussiana e crossover aritmético, simples, geométrico, esférico e heurístico do GENOCOP. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
                                                    FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
                                                    FinalCrossover);

Operator MutOp11 = new GenocopUniformMutation(mutationRate);
MutOp11.Sampler = samplers[1];

Operator MutOp12 = new GenocopNonUniformMutation(mutationRate,2,this);
MutOp12.Sampler = samplers[1];

Operator MutOp13 = new GenocopBoundaryMutation(mutationRate);
MutOp13.Sampler = samplers[1];

Operator MutOp14 = new GenocopGaussianMutation(mutationRate, this);
MutOp14.Sampler = samplers[1];

Operator CroOp11 = new GenocopArithmeticalCrossover(crossoverRate);
CroOp11.Sampler = samplers[1];

Operator CroOp12 = new GenocopSimpleCrossover(crossoverRate);
CroOp12.Sampler = samplers[1];

Operator CroOp13 = new GenocopGeometricalCrossover(crossoverRate);
CroOp13.Sampler = samplers[1];

Operator CroOp14 = new GenocopSphereCrossover(crossoverRate);
CroOp14.Sampler = samplers[1];

Operator CroOp15 = new GenocopHeuristicCrossover(crossoverRate, 100);
CroOp15.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento:

```
opsel1.addOperator(MutOp11, 1, 0);
opsel1.addOperator(MutOp12, 1, 0);
opsel1.addOperator(MutOp13, 1, 0);
opsel1.addOperator(MutOp14, 1, 0);
opsel1.addOperator(CroOp11, 1, 0);
opsel1.addOperator(CroOp12, 1, 0);
opsel1.addOperator(CroOp13, 1, 0);
opsel1.addOperator(CroOp14, 1, 0);
opsel1.addOperator(CroOp15, 1, 0);
```

O comando abaixo cria o operador que combina os indivíuos das 2 populações envolvidas nos problemas com o GENOCOP 3.

```
CrossoverSpecialOperator CroOp21 = new
                    GenocopIIIArithmeticalCrossover(crossoverRate);
CroOp21.Sampler = samplers[1];
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
GenocopEvolution evol2 = new GenocopEvolution(gensel1, repr1, "popr");
```

A primeira linha instancia a classe de reprodução, indicando os operadores genéticos escolhidos e o *steady-state*. A linha seguinte define o modelo de seleção dos genitores (no caso, por roleta), e passa o amostrador para o mesmo. Em seguida, instancia-se a classe que controla a evolução, passando como parâmetros a técnica de seleção e a reprodução.

A seguir, define-se a instância de avaliação:

```
evaluationFunc = new TesteEvaluation();
```

Tem-se a definição do decodificador que será utilizado pelo algoritmo genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial:

```
CoevolutionaryDecoder dec = new CoevolutionaryDecoder(evaluationFunc);
```

Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado e os melhores indivíduos da outra geração).

```
SelectBestFromOthers selOthers = new SelectBestFromOthers();
CoevolutionaryEvaluation eval = new CoevolutionaryEvaluation(dec,
                                        samplers[1], selOthers);
GenocopEvaluation eval2 = new GenocopEvaluation(dec, samplers[1],
                          selOthers, gensel1, "popr", 0.2, CroOp21);
```

O próximo passo consiste em instanciar o modo de inicialização da população. O modo de inicialização usado pelo GACOM em problemas com o GENOCOP 3 e aquele mostrado abaixo. Ele cria 1 indivíduo aleatório e os demais indivíduos na primeira geração são cópias deste criado. Isto ocorre pois pode ser computacionalmente inviável criar todos os indivíduos aleatoriamente, devido as dificuldades geradas pelas restrições.

```
InitOneIndividual indInit = new InitOneIndividual();
```

Em seguida, cria-se a população. A população recebe como parâmetros o número de indivíduos que a constituirão, uma string utilizada para identificá-la, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população.

```
Population pops = new Population(pop_size_, "pops", eval2,
                                                  evol2, this, indInit);
GenocopPopulation popr = new GenocopPopulation(pop_size_,
                            "popr", eval, evol1, this, indInit, pops);
```

No caso do GENOCOP 3 deve-se criar 2 populações. Uma de busca e outra de referência. Observe que pops está contida em popr. Mais informações podem ser obtidas em:

“Genetic Algorithms + Data Structures = Evolutionary Programs”, Zbigniew Michalewicz, capítulo 7.

As 2 linhas abaixo informam ao GA que o problema é de minimização.

```
pops.Maximization = false;
popr.Maximization = false;
```

Por último, instancia-se o modelo de trace, que guarda informações relativas à população.

```
ITrace trace_model = new Trace();

pops.TraceModel = trace_model;
popr.TraceModel = trace_model;
```

Após a criação da população, deve-se ainda adicionar o método de normalização linear, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new SimpleLinearNormalization(1,
PopulationSize);

pops.addBlock(2, norm);
popr.addBlock(2, norm);
```

Finalmente, deve-se preencher a população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população. Como o exemplo utiliza-se de duas populações distintas, tem-se duas formas de preencher a população.

```
fillPopulationsS(ref pops);
fillPopulationsR(ref popr, ilc1);

private void fillPopulationsS(ref Population population)
{
  Individual ind = null;
  GenocopSegment seg1 = null;
  GENOCOPRealGene gene = null;

  double min_value_clock = -10.0;
  double max_value_clock = 10.0;

  IneqLinearConstraints ilc1 = new IneqLinearConstraints("linear4.txt",
7);

  ind = new Individual(1);
  seg1 = new GenocopSegment(7, 10000, ilc1);
```

O valor 1000 acima informa a quantidade máxima de tentativas para o GA criar um indivíduo válido, o valor 7 se refere ao número de genes que o segmento possui.

```
    for (int j = 0; j < 7; j++)
    {
      gene = new GENOCOPRealGene(0, min_value_clock, max_value_clock);
      seg1.AddGene(gene);
    }

    ind.AddSegment(seg1);

    ind.Fitness = new RealFitness();
    ind.OriginalFitness = new RealFitness();
    ind.SpecieName = "clock";

    population.addIndividual(ind);
    for (int j = 0; j < PopulationSize - 2; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }

  private void fillPopulationsR(ref GenocopPopulation population)
  {
    Individual ind = null;
    GenocopSegment seg1 = null;
    GENOCOPRealGene gene = null;

    double min_value_clock = -10.0;
    double max_value_clock = 10.0;

    IneqLinearConstraints ilc1 = new IneqLinearConstraints("linear4.txt",
  7);

    G3NoLinearCons inlc1 = new G3NoLinearCons();

    ind = new Individual(1);
    seg1 = new GenocopSegment(7, 10000, ilc1, inlc1);

    for (int j = 0; j < 7; j++)
    {
      gene = new GENOCOPRealGene(0, min_value_clock, max_value_clock);
      seg1.AddGene(gene);
    }

    ind.AddSegment(seg1);

    ind.Fitness = new RealFitness();
    ind.OriginalFitness = new RealFitness();
    ind.SpecieName = "clock";

    population.addIndividual(ind);

    for (int j = 0; j < PopulationSize - 2; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado como parâmetro a quantidade de segmentos que esse possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Neste caso, como todo o cromossomo é real, o indivíduo possui apenas um segmento. Na criação do segmento é indicado por quantos genes este será composto. Após criar o segmento, adiciona-se os genes ao mesmo, cuidando para indicar os limites mínimos e máximos que o gene pode ter (neste caso, o intervalo [-10,10]). Feito isso, adiciona-se o segmento ao indivíduo. As duas linhas seguintes servem para indicar ao GACOM que tanto a avaliação (*OriginalFitness*) quanto a aptidão (*Fitness*) são valores reais.

Como o GENOCOP 3 utiliza-se de duas populações, uma com as restrições lineares e outra com, além destas, as não-lineares, é preciso defini-as. Para as restrições lineares, instancia-se a classe *IneqLinearConstraints*, passando como parâmetro um arquivo com os coeficientes de cada operador linear separados por espaço e um parâmetro correspondente à quantidade de incógnitas. Já para as restrições não lineares, criamos uma classe chamada *G3NoLinearCons*, a qual herda *IneqNoLinearConstraints* e sobrescreve o método *IsFeasible* com as restrições. Um exemplo dessa classe será apresentada mais a frente nesse tutorial.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método *Clone*.

Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Agora, deve-se adicionar a população recém-criada ao algoritmo genético. Isto pode ser feito através do seguinte comando:

```
addPopulation(popr);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition,* informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
SimpleGAConsoleOutput cvout = new SimpleGAConsoleOutput(this);
Output = cvout;
```

A classe utilizada para as restrições não lineares precisa herdar a classe *IneqNoLinearConstraints* e sobrescrever o método `IsFeasible()`. A implementação atual é apresentada abaixo:

```
using System;
using GACOM.Interfaces;
using GACOM.ConstraintsSpace;

public class G3NoLinearCons : IneqNoLinearConstraints
{
  public G3NoLinearCons()
  { }

  public override sealed bool IsFeasible(ISegment segment)
  {
    double[] x = new double[segment.TotalGenes];
    double y1, y2, y3, y4;
    for (int i = 0; i < segment.TotalGenes; i++)
    {
      x[i] = (double)segment.getGeneAt(i).Value;
    }

    y1 = 127 - 2*Math.Pow(x[0],2) - 3*Math.Pow(x[1],4) - x[2] -
4*Math.Pow(x[3],2) - 5*x[4];
    y2 = 282 - 7*x[0] - 3*x[1] - 10*Math.Pow(x[2],2) - x[3] + x[4];
    y3 = 196 - 23*x[0] - Math.Pow(x[1],2) - 6*Math.Pow(x[5],2) +
8*x[6];
    y4 = -4*Math.Pow(x[0],2) - Math.Pow(x[1],2) + 3*x[0]*x[1] -
2*Math.Pow(x[2],2) - 5*x[5] + 11*x[6];

    if ((y1 >= 0) && (y2 >= 0) && (y3 >= 0) && (y4 >= 0))
      return true;
    else
      return false;
  }
}
```

Além destas classes, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TesteEvaluation : FitnessFunction
{
```

O método `evaluate()` da classe *FitnessFuction* deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objs,
                          ref IIndividual individual, int generation)
{
```

A código completo desta classe é mostrado abaixo:

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.Interfaces;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
```

```
public class TesteEvaluation : FitnessFunction
{
  public TesteEvaluation()
  { }

  private const double PI = 3.141592654;

  public override Fitness evaluate(IGene[] objs,
                        ref IIndividual individual, int generation)

  {
    if (objs.Length == 7)
    {
      IGene gene1 = (IGene)objs[0];
      IGene gene2 = (IGene)objs[1];
      IGene gene3 = (IGene)objs[2];
      IGene gene4 = (IGene)objs[3];
      IGene gene5 = (IGene)objs[4];
      IGene gene6 = (IGene)objs[5];
      IGene gene7 = (IGene)objs[6];

      double x1, x2, x3, x4, x5, x6, x7;
      x1 = (double)gene1.Value;
      x2 = (double)gene2.Value;
      x3 = (double)gene3.Value;
      x4 = (double)gene4.Value;
      x5 = (double)gene5.Value;
      x6 = (double)gene6.Value;
      x7 = (double)gene7.Value;

      double r = G3(x1, x2, x3, x4, x5, x6, x7);

      individual.OriginalFitness = new RealFitness(r);

      for (int i = 0; i < individual.TotalSegments; i++)
      {
        if ((string)individual.getSegmentAt(i).Type=="GenocopSegment")
         ((GenocopSegment)individual.getSegmentAt(i)).FitnessValue =
                                      individual.OriginalFitness.Value;
      }

      individual.Fitness = new
                 RealFitness((double)individual.OriginalFitness.Value);

      return new RealFitness((double)individual.OriginalFitness.Value);
    }
    return null;
  }

  public double G3(double x1, double x2, double x3, double x4,
                                        double x5, double x6, double x7)
  {
    double d1;

    d1 = Math.Pow((x1-10),2) + 5*Math.Pow((x2-12),2) + Math.Pow(x3,4) +
3*Math.Pow((x4-11),2) + 10*Math.Pow(x5,6) + 7*Math.Pow(x6,2) +
Math.Pow(x7,4) - 4*x6*x7 - 10*x6 - 8*x7;
```

```
      return d1;
    }
  }
```

Para ilustrar, temos um programa que utiliza o GACOM com alguns parâmetros configurados:

```
using System;
using GACOM;

class TestingGA
{
  static void Main(string[] args)
  {
    TesteGA ga = new TesteGA("TesteGA", null);

    ga.Generations = 100;
    ga.PopulationSize = 400;
    ga.InitialCrossover = 0.8;
    ga.FinalCrossover = 0.65;
    ga.InitialMutation = 0.08;
    ga.FinalMutation = 0.8;
    ga.InitialSteadyState = 0.7;
    ga.FinalSteadyState = 0.3;
    ga.Experiments = 1;

    ga.exec();

    Console.ReadKey();
  }
}
```

Representação Baseada em Ordem

O exemplo deste tutorial será uma aplicação simples, utilizando a representação baseada em ordem. O objetivo deste exemplo é otimizar o problema do caixeiro viajante com restrição de precedência. As restrições de precedência, neste caso, é a obrigatoriedade do caixeiro passar por algumas cidades antes de chegar a outras, conforme será explicado mais a frente.

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto mais comum:

```
using GACOM.ConstraintsSpace;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.EvolutionProcess.CoevolutionaryIndividualSelection;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSpace.OrderBasedOperatorsSpace;
using GACOM.EvolutionProcess.RateSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.Interfaces;
using GACOM.OutputSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.StopConditionSpace;
using GACOM.Structures.GASpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using System;
using System.Collections;
```

Para o caso específico do caixeiro viajante apresentado aqui, deve-se ainda utilizar um pacote para a interface com o MatLab (EngMATLib):

```
using EngMATLib;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class TSPGA : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public TSPGA(string name, IStopCondition stop_condition)
    :
  base(name, stop_condition) { }
```

Agora, deve-se definir o método `Init()`:

```
public override void Init()
{
```

Agora defini-se o corpo deste método. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possivel criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;
p_random = new IRandom[2];
p_random[0] = new MRG32k3aRandGen();
p_random[1] = new LGMRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações.

Pode-se também definir o *steady-state*, caso seja necessária a utilização do mesmo. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Para isso, pode-se utilizar o comando abaixo:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
                  initial_steadystate_, final_steadystate_), false);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso, seleção por roleta) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, são usados operadores de mutação e *crossover* para cromossomos baseados em ordem. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
FinalCrossover);

Operator MutOp11 = new SwapMutation(mutationRate);
MutOp11.Sampler = samplers[1];

Operator MutOp12 = new PIMutation(mutationRate);
MutOp12.Sampler = samplers[1];

Operator CroOp11 = new PMXCrossover(crossoverRate);
CroOp11.Sampler = samplers[1];

Operator CroOp12 = new OXCrossover(crossoverRate);
CroOp12.Sampler = samplers[1];

Operator CroOp13 = new CXCrossover(crossoverRate);
CroOp13.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento:

```
opsel1.addOperator(MutOp11, 1, 0);
opsel1.addOperator(MutOp12, 1, 0);
opsel1.addOperator(CroOp11, 1, 0);
opsel1.addOperator(CroOp12, 1, 0);
opsel1.addOperator(CroOp13, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
```

A primeira linha instancia a classe de reprodução, indicando os operadores genéticos escolhidos e o *steady-state*. A linha seguinte define o modelo de seleção dos genitores (no caso, por roleta), e passa o amostrador para o mesmo. Em seguida, instancia-se a classe que controla a evolução, passando como parâmetros a técnica de seleção e a reprodução.

A seguir instancia-se a classe responsável pela restrições de precedência, indicando o caminho do arquivo xml que descreve as restrições. Este arquivo será explicado mais a diante.

```
PrecedenceConstraints pc = new PrecedenceConstraints("grafo.xml");
```

No caso deste exemplo, o arquivo estava localizado na pasta "F:\usr\Omar\Trabalho\Exemplo_tsp\TSP\bin\Debug" que é a pasta de debug do projeto de

execução “TSP”. Neste caso não foi necessário especificar o caminho. ATENÇÂO: Caso o arquivo esteja em uma outra pasta, o caminho deve ser especificado de forma completa.

```
evaluationFunc = new TSPConsEvaluation();
evaluationFunc.PrecConstraints = pc;
```

A seguir o código continua a consolidar uma série de definições.

```
Decoder dec = new Decoder(evaluationFunc);
Evaluation eval = new Evaluation(dec);
InitEachIndividual indInit = new InitEachIndividual();
Population pop = new Population(PopulationSize, "pop", eval, evol1,
this, indInit);
ITrace trace_model = new Trace();
pop.TraceModel = trace_model;
```

Na primeira linha acima, tem-se a definição do decodificador que será utilizado pelo algoritmo genético durante a avaliação. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. A população recebe como parâmetros o número de indivíduos que a constituem, uma string utilizada para identificá-la, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. Por último, instancia-se o modelo de trace, que guarda informações relativas à população.

Após a criação da população, deve-se ainda adicionar o método de normalização linear, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                     SimpleLinearNormalization(1, PopulationSize);
pop.addBlock(2, norm);
```

Para este exemplo, ainda é utilizado um método que mostra os resultados (grafo) a cada geração. Uma classe é criada para isto e o método necessita do caminho do código do matlab. (NÃO ESQUEÇA DE INFORMAR)

```
ShowBest sb = new ShowBest(TamCromo);
sb.PrecConstraints = pc;
pop.addBlock(2, sb);
```

Finalmente, deve-se preencher a população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população que será preenchida:

```
private void fillPopulations(ref Population population)
{
  Individual ind = null;
  OrderBasedSegment seg1 = null;
  OrderBasedGene gene = null;
```

```
    ArrayList list;
    list = new ArrayList();
    for (int i = 0; i < TamCromo; i++)
    {
      list.Add(i);
    }

    ind = new Individual(1);
    seg1 = new OrderBasedSegment(TamCromo, list);

    for (int j = 0; j < TamCromo; j++)
    {
      gene = new OrderBasedGene();
      seg1.AddGene(gene);
    }

    ind.AddSegment(seg1);

    ind.Fitness = new RealFitness();
    ind.OriginalFitness = new RealFitness();

    population.addIndividual(ind);

    for (int j = 0; j < PopulationSize - 1; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Para a criação do indivíduo é necessário passar o número de segmentos que o compõem. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Neste caso, como todo o cromossomo é baseado em ordem, o indivíduo possui apenas um segmento. Para o segmento, é necessário passar a quantidade de genes, que o compõem, como parâmetro. Após criar o segmento, adiciona-se os genes ao mesmo. Feito isso, adiciona-se o segmento ao indivíduo. As duas linhas seguintes servem para indicar ao GACOM que tanto a avaliação (*OriginalFitness*) quanto a aptidão (*Fitness*) são valores reais.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método *Clone*. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Agora, deve-se adicionar a população recém-criada ao algoritmo genético. Isto pode ser feito através do seguinte comando:

```
addPopulation(pop);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
Output = new SimpleGAConsoleOutput(this);
```

O código completo pode ser observado no arquivo TSPGA.cs que acompanha o exemplo do caixeiro viajante. Nesta classe ainda existem 2 outros métodos que são utilizados para mostrar os resultados da evolução. Observe que estes métodos precisam do caminho do arquivo XML que representa o grafo de precedência e o caminho do código do matlab.

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *PrecedenceFitnessFunction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TSPConsEvaluation : PrecedenceFitnessFunction
{
```

O método `evaluate()` da classe *PrecedenceFitnessFunction* deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objects,
                        ref IIndividual individual, int generation)
{
```

A código completo desta classe é encontrado no arquivo TSPConsEvaluation.cs e é importante observar os comentário contidos no código:

```
using EngMATLib;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
using System.Collections;
using System;

public class TSPConsEvaluation : PrecedenceFitnessFunction
{
  private EngMATAccess _eng_mat_ = null;

  public TSPConsEvaluation()
  {
    _eng_mat_ = new EngMATAccess();
    _eng_mat_.SetVisible(false);

    string command = @"cd('\')";
    string ret = _eng_mat_.EvaluateAsString(command);

    // Este é o caminho onde está o código do MATLAB
    command = @"cd('D:\usr\leandro\Projects\GACOM-
Tutoriais\TSP\eval')";
    ret = _eng_mat_.EvaluateAsString(command);
  }
```

```
  public override Fitness evaluate(IGene[] objects,
                       ref IIndividual individual, int generation)
  {
    try
    {
      string command;
      string ret;
      string vetOrder;
      double eval;
      int tam_cromo;
      ArrayList solution;

      tam_cromo = 20; // Tamanho do cromossomo

      // Gera solução real através do grafo de precedência
      solution = CreateSolution(objects);

      // Pega vetor com as cidades
      vetOrder = @"[";
      for (int i = 0; i < tam_cromo; i++)
      {
        int v;
        IGene g;

        g = (IGene)solution[i];
        v = 0;
        v = (int)g.Value;
        v += 1;
        vetOrder += v.ToString() + @" ";
      }

      vetOrder += @"]";
      command = @"clear eval";
      ret = _eng_mat_.EvaluateAsString(command);

      // Cria comando do matlab
      command = @"eval = tsp(" + vetOrder + ", 0)";

      // Faz chamada do matlab
      ret = _eng_mat_.EvaluateAsString(command);

      // Pega avaliação
      eval = _eng_mat_.GetMatrix("eval").AsVector()[0];

      individual.OriginalFitness = new RealFitness(eval);
      individual.Fitness = new RealFitness(eval);

      return individual.Fitness;
    }
    catch (Exception e)
    {
      throw new Exception("TSPConsEvaluation: (evaluate) " +
e.Message);
    }
  }

  public ArrayList CreateSolution(IGene[] objs)
  {
    ArrayList objects = new ArrayList();
```

```
      for (int i = 0; i < objs.Length; i++)
      {
        objects.Add(objs[i]);
      }

      int cont = 0;
      int contgen = 0;
      int v = 0;
      bool noConst = true;
      ArrayList solution = new ArrayList();
      double[,] matrix =
  (double[,])PrecConstraints.GetPrecedenceMatrix().Clone();
      int numElements = objects.Count;

      while (cont < objects.Count)
      {
        noConst = true;
        IGene g;
        g = (IGene)objects[cont];
        v = (int)g.Value;
        for (int i = 0; i < numElements; i++)
        {
          if (matrix[v, i] == 1.0)
          {
            noConst = false;
            break;
          }
        }

        if (noConst)
        {
          solution.Add(((IGene)objects[cont]).Clone());
          contgen += 1;

          for (int j = 0; j < numElements; j++)
          {
            if (matrix[j, v] == 1.0)
              matrix[j, v] = 0.0;
          }

          objects.RemoveAt(cont);
          cont = 0;
        }
        else
          cont += 1;
      }
      return solution;
    }
  }
```

Por fim, precisamos definir o programa principal que irá chamar o GAcom. A seguir temos um exemplo de programa que chama o GACOM:

```
  using System;

  namespace Teste
  {
    class SCH1Run
```

```
  {
    static void Main(string[] args)
    {
      TSPGA ga = new TSPGA("TSPGA", null);

      ga.Generations = 180;
      ga.PopulationSize = 100;
      ga.InitialCrossover = 0.9;
      ga.FinalCrossover = 0.1;
      ga.InitialMutation = 0.08;
      ga.FinalMutation = 0.8;
      ga.InitialSteadyState = 0.9;
      ga.FinalSteadyState = 0.3;
      ga.Experiments = 1;

      ga.exec();
    }
  }
}
```

Arquivo XML com as restrições de precedência

O Arquivo de restrições deve seguir o seguinte formato:

```
<?xml version = "1.0"?>
<grafo_precedencia>

  <elements>20</elements>

  <no name = "0">
    <precedence>1</precedence>
    <precedence>2</precedence>
    <precedence>18</precedence>
    <precedence>17</precedence>
    <precedence>12</precedence>
  </no>

  <no name = "14">
    <precedence>4</precedence>
    <precedence>13</precedence>
    <precedence>11</precedence>
    <precedence>8</precedence>
    <precedence>19</precedence>
  </no>

  <no name = "12">
    <precedence>2</precedence>
    <precedence>1</precedence>
    <precedence>9</precedence>
  </no>

  <no name = "4">
    <precedence>0</precedence>
    <precedence>13</precedence>
    <precedence>11</precedence>
  </no>
```

```
</grafo_precedencia>
```

Onde o tag *elements* indica o tamanho da matriz de precedência. No caso deste exemplo o caixeiro viajante percorre 20 cidade, logo a matriz tem tamanho 20x20.

O tag no indica uma cidade e o tag precedence, filho do tag no, indica as restrições de precedência. Por exemplo, segundo o primeiro tag no, cujo name é igual a “0”, o caixeiro deve passar nas cidades representadas por 1, 2, 18, 17 e 12 antes de passar pela cidade representada por “0”.

Os genes devem possuir os seguintes valores: [0 n-1], onde n é o tamanho do cromossomo. O usuário deve tomar cuidado ao mapear a solução final.

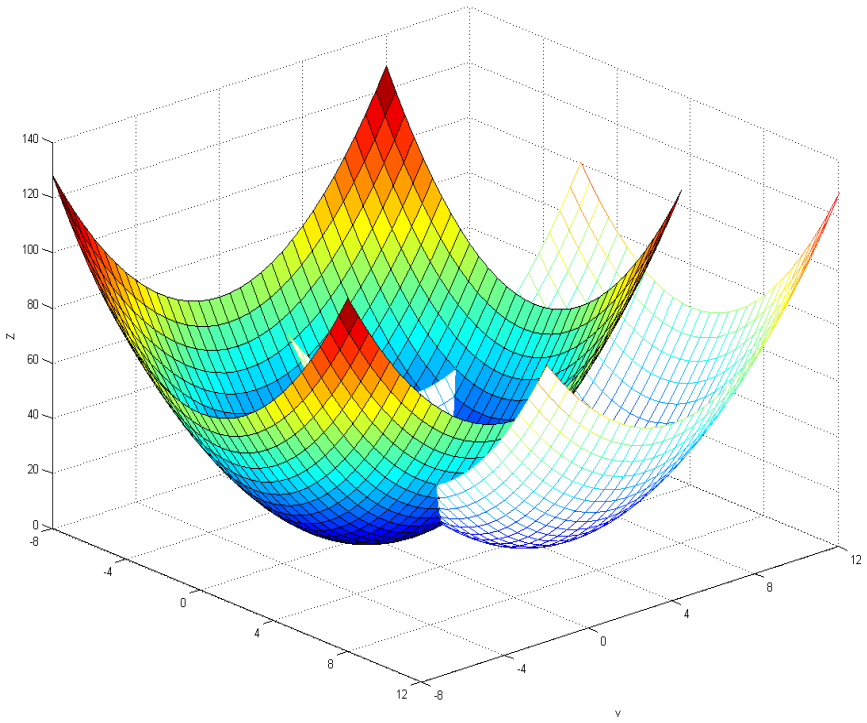
Exemplo:

No caso do caixeiro viajante utilizado aqui, o cromossomo tem tamanho 20, logo os valores no cromossomo vão de 0 a 19. Porém os valores das cidades no MATLAB vai de 1 a 20, logo deve haver uma decodificação deste cromossomo, somando 1 ao valor do gene no cromossomo.

Com isso, a cidade “0” no arquivo do grafo acima aparece como cidade “1” na solução final.

Problemas Muiti-Objetivos (Distância ao Alvo)

Para ilustrar um problema com objetivos múltiplos, é apresentado a minimização dos objetivos $z_1 = x^2 + y^2$ e $z_2 = (x-4)^2 + (y-4)^2$ com $x, y \in [-100,100]$ os quais possuem pontos ótimos em (0,0) e (4,4) respectivamente. O peso dos dois objetivos é o mesmo, logo o ótimo global encontra-se em (2,2). A figura abaixo ilustra os dois planos que representam o problema.



A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto utilizado:

```
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSpace.RealOperatorsSpace;
using GACOM.EvolutionProcess.RateSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.Interfaces;
using GACOM.OutputSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.StopConditionSpace;
using GACOM.Structures.GASpace;
using GACOM.Structures.GeneSpace;
```

```
using GACOM.Structures.IndividualSpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class SCH1GA : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public SCH1GA(string name, IStopCondition stop_condition)
  :
base(name, stop_condition) {}
```

Agora, inicia-se a construção do corpo do método construtor. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;

p_random = new IRandom[2];
p_random[0] = new LGMRandGen();
p_random[1] = new LGMRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Pode-se também definir o *steady-state*, caso seja necessária a utilização do mesmo. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Para isso, pode-se utilizar o comando abaixo:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
                   initial_steadystate_, final_steadystate_), true);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso, seleção or roleta) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, como os cromossomos são reais, são usados diversos operadores, entre eles os operadores de mutação uniforme, não uniforme e crossover aritmético. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
                                                    FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
                                                    FinalCrossover);

Operator CroOp1 = new ArithmeticalCrossover(crossoverRate);
CroOp1.Sampler = samplers[1];

Operator MutOp1 = new RealUniformMutation(mutationRate);
MutOp1.Sampler = samplers[1];

Operator MutOp2 = new SimpleNonUniformMutation(mutationRate,0.5,this);
MutOp2.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento só há um segmento, todos os operadores serão aplicados no mesmo segmento:

```
opsel1.addOperator(MutOp1, 1, 0);
opsel1.addOperator(MutOp2, 1, 0);
opsel1.addOperator(CroOp1, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
SCH1Evaluation evaluationFunc = new SCH1Evaluation ();
Decoder dec = new Decoder(evaluationFunc);
Evaluation eval = new Evaluation(dec);
InitEachIndividual indInit = new InitEachIndividual();
Population pop = new Population(pop_size_, "pop", eval,
                                              evol1, this, indInit);
ITrace trace_model = new Trace();
pop.TraceModel = trace_model;
pop.Maximization = false;
```

A primeira linha instancia a classe de reprodução, indicando os operadores genéticos escolhidos e o *steady-state*. A linha seguinte define o modelo de seleção dos genitores (no caso, por roleta), e passa o amostrador para o mesmo. Em seguida, instancia-se a classe que controla a evolução, passando como parâmetros a técnica de seleção e a reprodução. Na linha seguinte, tem-se a definição do decodificador que será utilizado pelo algoritmo

genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. A população recebe como parâmetros a quantidade de indivíduos que a constituem (tamanho da população), uma string utilizada para identificá-la, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. Por último, instancia-se o modelo de trace, que guarda informações relativas à população, e defini-se o problema como de minimização.

Após a criação da população, deve-se ainda adicionar o método de normalização linear e adicioná-la a um bloco, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                        SimpleLinearNormalization(1, PopulationSize);
pop.addBlock(2, norm);
```

Finalmente, deve-se preencher a população com os indivíduos iniciais. O trecho do código abaixo mostra como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
private void fillPopulations(ref Population population)
{
  Individual ind = null;
  Segment seg1 = null;
  Gene gene = null;

  double min_value_clock = -100.0;
  double max_value_clock = 100.0;

  ind = new Individual(1);
  seg1 = new Segment(TamCromo);

  for (int j = 0; j < TamCromo; j++)
  {
    gene = new RealGene(0, min_value_clock, max_value_clock);
    seg1.AddGene(gene);
  }

  ind.AddSegment(seg1);

  ind.Fitness = new RealFitness();
  ind.OriginalFitness = new MultiRealFitness(2);
  ind.MultiOriginalFitness = new RealFitness();

  population.addIndividual(ind);

  for (int j = 0; j < PopulationSize - 1; j++)
    population.addIndividual((IIndividual)ind.Clone());

}
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado o número de segmentos que este possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Neste caso, como todo o cromossomo é binário, o indivíduo possui apenas um segmento. Para a criação deste, é passado o números de genes que o constituirão. Após criar o segmento, adiciona-se os genes ao mesmo. Feito isso, adiciona-se o segmento ao indivíduo. As três linhas seguintes servem para indicar ao GACOM que a aptidão (*Fitness*) é real, enquanto a avaliação (*OriginalFitness*) é do tipo multi-objetivo com 2 objetivos a serem otimizados, e cada um desses objetivos possui uma avaliação (*MultiOriginalFitness*) real.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método Clone. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Agora, deve-se adicionar a população recém-criada ao algoritmo genético. Isto pode ser eito através do seguinte comando:

```
addPopulation(pop);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
SimpleGAConsoleOutput cvout = new SimpleGAConsoleOutput(this);
```

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class TesteEvaluation : FitnessFunction
{
```

O método *evaluate* da classe FitnessFuction deve ser reescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objs,
    ref IIndividual individual, int generation)
{
```

O código completo desta classe é mostrado abaixo:

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
```

```
using GACOM.Interfaces;
using GACOM.EvaluationProcess.FitnessSpace;
using System.Collections;
using System.IO;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
using MULTICOM.MethodsSpace;

public class SCH1Evaluation : FitnessFunction
{
  TargetDistance method;

  public SCH1Evaluation()
  {
    double[] user = { 0.0, 0.0 };
    int p = 2;
    method = new TargetDistance(user, p);
  }

  public override Fitness evaluate(IGene[] objs, ref IIndividual[]
individuals, IIndividual otherPopIndividual, Fitness[] indFitness, int
generation)
  {
    int a;

    if (generation != 0)
      a = 0;

    ////////////////////////////////
    // Distância ao Alvo
    ////////////////////////////////
    int numberEvaluated = indFitness.Length;

    double[] fit = new double[2];
    double[] fitnessTotal = new double[1];

    IGene gene1 = null;
    IGene gene2 = null;

    for (int i = 0; i < individuals.Length; i++)
    {
      IIndividual ind =(IIndividual)(individuals[i]).Clone();

      if (!ind.Evaluated)
      {
        gene1 = (IGene)objs[2 * i];
        gene2 = (IGene)objs[2 * i + 1];

        // Avaliação do objetivo 1
        fit[0] = f1((double)gene1.Value, (double)gene2.Value);

        // Avaliação do objetivo 2
        fit[1] = f2((double)gene1.Value, (double)gene2.Value);

        fitnessTotal = method.FitnessEvaluation(fit);

        (individuals[i]).OriginalFitness = new MultiRealFitness(fit);
        (individuals[i]).Fitness = new RealFitness(fitnessTotal[0]);
        (individuals[i]).MultiOriginalFitness = new
RealFitness(fitnessTotal[0]);
```

```
      }
    }
    return null;
  }

  private double f1(double x, double y)
  {
    return Math.Pow(x, 2) + Math.Pow(y, 2);
  }

  private double f2(double x, double y)
  {
    return Math.Pow((x - 4), 2) + Math.Pow((y - 4), 2);
  }
}
```

Para a avaliação do multi-objetivo, é utilizada a biblioteca MULTICOM, desenvolvida pelo ICA. Através dela, utilizou-se nesse exemplo o método multi-objetivo de distância ao alvo. O método de minimização de energia será exemplificado no próximo tutorial. Então deve-se definir o *fitness* do indivíduo (caso ainda não tenha sido avaliado) como o resultado desse método. O *OriginalFitness* continua sendo a avaliação original de cada objetivo e o *Fitness* consiste da aptidão retornada pelo método de multiobjetivo que será normalizada para ser utilizada na seleção do indivíduo. Agora, além desses *fitness* que já eram utilizados, teremos o *MultiOriginalFitness*, o qual contém a aptidão retornada pelo método multiobjetivo utilizado.

Por fim, precisamos definir o programa principal que irá chamar o GAcom. A seguir temos um exemplo de programa que chama o GACOM:

```
using System;

namespace Multi_Objetivo
{
  class SCH1Run
  {
    static void Main(string[] args)
    {
      SCH1GA ga = new SCH1GA("SCH1", null);

      ga.Generations = 300;
      ga.PopulationSize = 100;
      ga.InitialCrossover = 0.8;
      ga.FinalCrossover = 0.5;
      ga.InitialMutation = 0.1;
      ga.FinalMutation = 0.3;
      ga.InitialSteadyState = 0.2;
      ga.FinalSteadyState = 0.2;
      ga.Experiments = 2;

      ga.exec();
    }
  }
}
```

Problemas Muiti-Objetivos (Minimização de Energia)

Para exemplificar a utilização do método de minimização de energia, utilizaremos o mesmo problema descrito anteriormente. A estrutura do GA segue a mesma, a única diferença estará no método *evaluate* da classe *FitnessFuction* que deve ser reescrito nesta classe. Conforme mostrado abaixo:

```
using System;
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.Interfaces;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
using MULTICOM.MethodsSpace;

namespace Multi_Objetivo
{
  public class SCH1Evaluation2 : FitnessFunction
  {
    MinimizeEnergy method;

    public SCH1Evaluation2()
    {
      //Minimização de Energia
      method = new MinimizeEnergy(2);
    }

    public override Fitness evaluate(IGene[] objs, ref IIndividual[]
individuals, IIndividual otherPopIndividual, Fitness[] indFitness, int
generation)
    {
      int a;

      if (generation != 0)
      a = 0;

      ////////////////////////////////
      // Minimização de Energia
      ////////////////////////////////

      int numberEvaluated = indFitness.Length;

      double[] user = { 0.0, 0.0 };
      double alpha = 0.7;

      double[] fitnessTotal = new double[individuals.Length];
      double[,] fit = new double[individuals.Length, 2];
      double[,] fitness = new double[individuals.Length, 2];

      IGene gene1 = null;
      IGene gene2 = null;

      for (int i = 0; i < indFitness.Length; i++)
      {
        fit[i, 0] =
((double[])((MultiRealFitness)indFitness[i]).Value)[0];
```

```
            fit[i, 1] =
    ((double[])((MultiRealFitness)indFitness[i]).Value)[1];
          }

          for (int i = indFitness.Length; i < individuals.Length; i++)
          {
            gene1 = (IGene)objs[2 * i];
            gene2 = (IGene)objs[2 * i + 1];

            // objetivo 1
            fit[i, 0] = f1((double)gene1.Value, (double)gene2.Value);
            // objetivo 2
            fit[i, 1] = f2((double)gene1.Value, (double)gene2.Value);
          }

          fitnessTotal = method.FitnessEvaluation(fit, user, alpha,
    generation);

          double[] fitnessAux = new double[2];
          for (int j = 0; j < individuals.Length; j++)
          {
            fitnessAux[0] = fit[j, 0];
            fitnessAux[1] = fit[j, 1];
            if (j >= indFitness.Length)
              ((IIndividual)(individuals[j])).OriginalFitness = new
    MultiRealFitness(fitnessAux);
             ((IIndividual)(individuals[j])).Fitness = new
    RealFitness(fitnessTotal[j]);
             ((IIndividual)(individuals[j])).MultiOriginalFitness = new
    RealFitness(fitnessTotal[j]);

          }
        return null;
      }

      private double f1(double x, double y)
      {
        return Math.Pow(x, 2) + Math.Pow(y, 2);
      }

      private double f2(double x, double y)
      {
        return Math.Pow((x - 4), 2) + Math.Pow((y - 4), 2);
      }
    }
}
```

Deve-se observar que, para esse caso, mesmo se o indivíduo já estiver sido avaliado, é preciso reavaliá-lo, uma vez que o peso de cada objetivo muda dinamicamente de acordo com a média da população.

Problemas Co-Evolucionários

Para ilustrar um problema co-evolucionário, suponhamos que seja necessário utilizar cinco aviões para uma missão humanitária: distribuir comida para o maior numero de cidades (num universo de 25) afetadas durante uma guerra. Para isso há a disponibilidade de 15 aviões, cada um com uma autonomia diferente e todos devem ser encarregados de suprir um número de cidades que seja múltiplo de 5. Sendo assim, duas populações devem ser evoluídas, uma população para calcular a menor rota entre as cidades e outra para escolher a melhor combinação de aviões para essas rotas.

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto utilizado:

```
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.EvaluationProcess;
using GACOM.EvolutionProcess.CoevolutionaryIndividualSelection;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.OperatorSpace.OrderBasedOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSpace.IntegerOperatorsSpace;
using GACOM.EvolutionProcess.RateSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.Interfaces;
using GACOM.OutputSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.StopConditionSpace;
using GACOM.Structures.GASpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using System.Collections;
using System;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class SCH1GA : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public CoevGA(string name, IStopCondition stop_condition)
  :
base(name, stop_condition) {}
```

Agora, inicia-se a construção do corpo do método construtor. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;

p_random = new IRandom[2];
p_random[0] = new LGMRandGen();
p_random[1] = new LGMRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Pode-se também definir o *steady-state*, caso seja necessária a utilização do mesmo. Este recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Nesse exemplo, duas intancias do steady state são utilizadas, uma para cada população:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
                  initial_steadystate_, final_steadystate_), false);

SteadyState eli2 = new SteadyState(new LinearRate(this,
                  initial_steadystate_, final_steadystate_), false);
```

O próximo passo consiste em definir a técnica de seleção de operadores (no caso usamos um operador para cada população) e o amostrador associado a ela:

```
RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];

RouletteOperatorSelection opsel2 = new RouletteOperatorSelection();
opsel2.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, como os cromossomos da população de cidades são baseados em ordem, enquanto que os cromossomos da população de aviões são inteiros. Sendo assim, é preciso utilizar operadores distintos para cada uma das populações. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
                                                    FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
                                                    FinalCrossover);

// Populacao de cidades
Operator MutOp11 = new SwapMutation(mutationRate);
MutOp11.Sampler = samplers[1];

Operator MutOp12 = new PIMutation(mutationRate);
MutOp12.Sampler = samplers[1];

Operator CroOp11 = new PMXCrossover(crossoverRate);
CroOp11.Sampler = samplers[1];

Operator CroOp12 = new OXCrossover(crossoverRate);
CroOp12.Sampler = samplers[1];

Operator CroOp13 = new CXCrossover(crossoverRate);
CroOp13.Sampler = samplers[1];

// Populacao de avioes
Operator Cro21 = new SinglePointIntegerCrossover(crossoverRate);
Cro21.Sampler = samplers[1];

Operator Cro22 = new TwoPointIntegerCrossover(crossoverRate);
Cro22.Sampler = samplers[1];

Operator Cro23 = new IntegerUniformCrossover(crossoverRate);
Cro23.Sampler = samplers[1];

Operator Mut21 = new IntegerUniformMutation(mutationRate);
Mut21.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento cada população possui apenas um segmento, todos os operadores serão aplicados no mesmo segmento, porém deve-se ter cuidado para atribuir os operadores corretos para as respectivas populações:

```
opsel1.addOperator(MutOp11, 1, 0);
opsel1.addOperator(MutOp12, 1, 0);
opsel1.addOperator(CroOp11, 1, 0);
opsel1.addOperator(CroOp12, 1, 0);
opsel1.addOperator(CroOp13, 1, 0);

opsel2.addOperator(Cro21, 1, 0);
opsel2.addOperator(Cro22, 1, 0);
opsel2.addOperator(Cro23, 1, 0);
opsel2.addOperator(Mut21, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
Reproduction repr2 = new Reproduction(opsel2, eli2);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
GenitorSelection gensel2 = new RouletteGenitorSelection(samplers[1]);
```

```
Evolution evol1 = new Evolution(gensel1, repr1);
Evolution evol2 = new Evolution(gensel2, repr2);
SelectBestFromOthers selectmethod = new SelectBestFromOthers();
CoevEvaluation evaluationFunc = new CoevEvaluation();
CoevolutionaryDecoder dec = new CoevolutionaryDecoder(evaluationFunc);
CoevolutionaryEvaluation eval = new CoevolutionaryEvaluation(dec,
samplers[1], selectmethod);
InitEachIndividual indInit = new InitEachIndividual();

// População de referência
Population pop_cities = new Population(PopulationSize, "Ordem das
Cidades", eval, evol1, this, indInit);

// População de referência
Population pop_planes = new Population(PopulationSize, "Avioes", eval,
evol2, this, indInit);
ITrace trace_model = new Trace();
pop_cities.TraceModel = trace_model;
pop_planes.TraceModel = trace_model;
pop_cities.Maximization = true;
pop_planes.Maximization = true;
```

As duas primeiras linhas instanciam a classe de reprodução de cada uma população, indicando os operadores genéticos escolhidos e o *steady-state*. As duas próximas linhas definem os modelos de seleção dos genitores (no caso, por roleta), e passam o amostrador para os mesmos. Em seguida, instanciam-se as classes que controlam a evolução, passando como parâmetros a técnica de seleção e a reprodução. Então é definido o método de seleção dos indivíduos da outra população para serem usados na avaliação da população corrente. Na linha seguinte, tem-se a definição do decodificador que será utilizado pelo algoritmo genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado, um amostrador e o método de seleção dos indivíduos da outra população). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. As populações recebem como parâmetros a quantidade de indivíduos que as constituem (tamanho da população), uma string utilizada para identificação, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. Por último, instancia-se o modelo de trace, que guarda informações relativas à cada população, e defini-se o problema como de maximização para ambas as populações por se tratar de um problema de simbiose.

Após a criação da população, deve-se ainda adicionar o método de normalização linear e adicioná-la a um bloco, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                        SimpleLinearNormalization(1, PopulationSize);
pop_cities.addBlock(2, norm);
pop_planes.addBlock(2, norm);
```

Finalmente, deve-se preencher as populações com os indivíduos iniciais. Os trechos de códigos abaixo mostram como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
private void fill_cities_population(ref Population population)
{
  Individual ind = null;
  OrderBasedSegment seg1 = null;
  OrderBasedGene gene = null;
  int TamCromo = 25; // Tamanho do cromossomo (número de cidades)
  ArrayList list;
  list = new ArrayList();

  for (int i = 0; i < TamCromo; i++)
  {
    list.Add(i);
  }

  ind = new Individual(1);
  seg1 = new OrderBasedSegment(TamCromo, list);
  for (int j = 0; j < TamCromo; j++)
  {
    gene = new OrderBasedGene();
    seg1.AddGene(gene);
  }
  ind.AddSegment(seg1);

  ind.Fitness = new RealFitness();
  ind.OriginalFitness = new RealFitness();

  population.addIndividual(ind);

  for (int j = 0; j < PopulationSize - 1; j++)
  {
    population.addIndividual((IIndividual)ind.Clone());
  }
}
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado o número de segmentos que este possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Nesse caso, como na população de cidades todo o cromossomo é baseado em ordem, o indivíduo possui apenas um segmento. Para a criação deste, é passado o números de genes que o constituirão. Após criar o segmento, adiciona-se os genes ao mesmo. Feito isso, adiciona-se o segmento ao indivíduo. As três linhas seguintes servem para indicar ao GACOM que a aptidão (*Fitness*) e a avaliação (*OriginalFitness*) são reais.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método Clone. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Para a população de aviões utilizados a principal diferença é que o gene agora é inteiro e deve-se passar o número máximo e mínimo que cada gene pode atingir.

```
private void fill_planes_population(ref Population population)
{
  Individual ind = null;
  Segment seg1 = null;
  IntegerGene gene = null;
  int TamCromo = 5; // 5 avioes escolhidos
  int min = 0;
  int max = 14;

  ind = new Individual(1);
  seg1 = new Segment(TamCromo);

  for (int j = 0; j < TamCromo; j++)
  {
    gene = new IntegerGene(min,max);
    seg1.AddGene(gene);
  }

  ind.AddSegment(seg1);

  ind.Fitness = new RealFitness();
  ind.OriginalFitness = new RealFitness();

  population.addIndividual(ind);

  for (int j = 0; j < PopulationSize - 1; j++)
  {
    population.addIndividual((IIndividual)ind.Clone());
  }
}
```

Agora, deve-se criar um vetor de populações com tamanho igual ao número de populações que devem co-evoluir e em seguida adicionar as populações recém-criadas ao algoritmo genético. Isto pode ser feito através do seguinte comando:

```
this.p_pops = new IPopulation[2];
addPopulation(pop_cities);
addPopulation(pop_planes);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
CoevolutionaryConsoleOutput cvout = new
                                    CoevolutionaryConsoleOutput(this);
```

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class CoevEvaluation : FitnessFunction
{
```

O método *evaluate* da classe FitnessFuction deve ser sobrescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objects,
    ref IIndividual individual, int generation)
{
```

O código completo desta classe é mostrado abaixo:

```
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
using System.Collections;
using System;
using System.IO;

namespace Coev
{
  public class CoevEvaluation : FitnessFunction
  {
    private double[] planesAuto;
    private double[,] cities;

    public CoevEvaluation()
    {
      planesAuto = new double[15];
      cities = new double[26, 2];

      //carrega os pesos dos avioes
      StreamReader sr = File.OpenText(@"D:\usr\leandro\Projects\GACOM-
Tutoriais\TSP-mod\eval\plane.txt");

      string line;
      int i = 0;

      while (((line = sr.ReadLine()) != null) && (i <
planesAuto.GetLength(0)))
      {
        line = line.Trim();
        string[] word;
        if (line.Length > 0)
        {
          string[] space = { "\t", " " };
          word = line.Split(space,
                      StringSplitOptions.RemoveEmptyEntries);

          planesAuto[i++] = Convert.ToDouble(word[0]);
        }
      }
      sr.Close();
```

```
      //carrega as localizacoes das cidades
      sr = File.OpenText(@"D:\usr\leandro\Projects\GACOM-Tutoriais\TSP-
mod\eval\city.txt");

      i = 0;

      while (((line = sr.ReadLine()) != null) && (i <
cities.GetLength(0)))
      {
        line = line.Trim();
        string[] word;
        if (line.Length > 0)
        {
          string[] space = { "\t", " " };
          word = line.Split(space,
                            StringSplitOptions.RemoveEmptyEntries);
          cities[i, 0] = Convert.ToDouble(word[0]);
          cities[i++, 1] = Convert.ToDouble(word[1]);
        }
      }
      sr.Close();
    }

    public override Fitness evaluate(IGene[] objects, ref IIndividual
individual, int generation)
    {
      try
      {
        double next; // distancia euclidiana para a proxima cidade
        double home; // distancia euclidiana para voltar para a base
        int totalCities = 0;
        double autonomia;
        int[] cityIndex = new int[25];
        int[] avioes = new int[5];

        // armazena os elemntos referentes aos avioes e as cidades
        int posa = 0;
        int posc = 0;

        for (int i = 0; i < objects.Length; i++)
        {
          if (objects[i] is IntegerGene)
          {
            avioes[posa] = (int)objects[i].Value;
            posa++;
          }
          else
          {
            cityIndex[posc] = (int)objects[posc].Value;
            posc++;
          }
        }

        ArrayList solution = new ArrayList();
        ArrayList item;
        int cont;
```

```
        for (int i = 0; i < avioes.Length; i++)
        {
          cont = 0;
          foreach (object o in solution)
          {
            if ((int)(((ArrayList)o)[0]) == avioes[i])
            {
              for (int j = 0; j < 5; j++)
              {
                ((ArrayList)o).Add(cityIndex[(i * 5) + j]);
              }
              cont = 1;
              break;
            }
          }
          if (cont == 0)
          {
            item = new ArrayList();
            item.Add(avioes[i]);
            for (int j = 0; j < 5; j++)
            {
              item.Add(cityIndex[(i * 5) + j]);
            }
            solution.Add(item);
          }
        }

        foreach (object o in solution)
        {
          autonomia = planesAuto[(int)((ArrayList)o)[0]];

          // distancia para a primeira cidade
          next = Math.Pow(cities[(int)((ArrayList)o)[1] + 1, 0] -
cities[0, 0], 2);
          next += Math.Pow(cities[(int)((ArrayList)o)[1] + 1, 1] -
cities[0, 1], 2);
          next = Math.Sqrt(next);

          home = next;

          if (autonomia >= (next + home))
          {
            totalCities++;
            autonomia -= next;
          }
          else continue;

          for (int i = 1; i < ((ArrayList)o).Count - 1; i++)
          {
            // distancia da proxima cidade
            next = Math.Pow((cities[(int)((ArrayList)o)[i] + 1, 0] -
cities[(int)((ArrayList)o)[i + 1] + 1, 0]), 2);
            next += Math.Pow((cities[(int)((ArrayList)o)[i] + 1, 1] -
cities[(int)((ArrayList)o)[i + 1] + 1, 1]), 2);
            next =  Math.Sqrt(next);

            // distancia para voltar da proxima cidade para base
            home = Math.Pow((cities[(int)((ArrayList)o)[i + 1] + 1, 0]
```

```
- cities[0, 0]), 2);
            home += Math.Pow((cities[(int)((ArrayList)o)[i + 1] + 1, 1]
- cities[0, 1]), 2);
            home =  Math.Sqrt(home);

    // verifica se tem combustivel para ir pra proxima cidade e voltar

            if (autonomia >= (next + home))
            {
              totalCities++;
              autonomia -= next;
            }
            else break;
          }
        }

        individual.OriginalFitness = new
                        RealFitness(Convert.ToDouble(totalCities));
        individual.Fitness = new
                        RealFitness(Convert.ToDouble(totalCities));

        return individual.Fitness;
      }
      catch (Exception e)
      {
        throw new Exception("CoevEvaluation: (evaluate) " + e.Message);
      }
    }
  }
}
```

Para a avaliação do co-evolucionáro, é preciso tratar os objetos de entrada uma vez que esses intercalam de acordo com a população que está sendo avaliada, para isso é feita a checagem de se o gene é inteiro ou real, podendo saber qual a população a qual aquele gene pertence. Após a execução da avaliação (que no caso é o número de cidades percorridas). deve-se definir o *fitness* do indivíduo como o resultado desse método. O *OriginalFitness* continua sendo a avaliação original de cada objetivo e o *Fitness* consiste da aptidão que será normalizada para ser utilizada na seleção do indivíduo.

Por fim, precisamos definir o programa principal que irá chamar o GAcom. A seguir temos um exemplo de programa que chama o GACOM:

```
using System;

namespace Coev
{
  class Program
  {
    static void Main(string[] args)
    {
      CoevGA ga = new CoevGA("TSPGA", null);

      ga.Generations = 100;
      ga.PopulationSize = 60;
      ga.InitialCrossover = 0.85;
      ga.FinalCrossover = 0.45;
```

```
            ga.InitialMutation = 0.08;
            ga.FinalMutation = 0.8;
            ga.InitialSteadyState = 0.6;
            ga.FinalSteadyState = 0.4;
            ga.Experiments = 1;

            ga.exec();

            Console.ReadKey();
        }
    }
}
```

GA Quântico

A fim de instruir o usuário no uso do GA Quântico, vamos voltar a aplicação da F6. O objetivo deste exemplo é maximizar a função F6, definida pela equação (1) e representada graficamente pela figura 1.

$$F6(x,y) = 0.5 - \frac{\left(\sin\sqrt{x^2+y^2}\right)^2 - 0.5}{1.0 + 0.001\left(x^2+y^2\right)^2} \qquad (1)$$

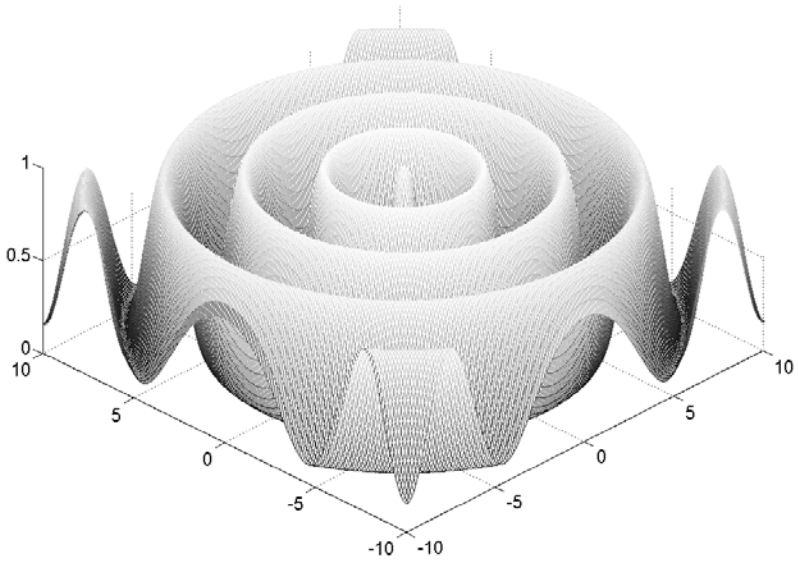


Figura 1. Representação gráfica da função F6.

A primeira coisa que deve ser feita ao se usar o GACOM é explicitar quais os pacotes que se deseja utilizar. O código abaixo mostra o conjunto utilizado:

```
using System;
using System.Collections;
using GACOM.Interfaces;
using GACOM.Structures.GASpace;
using GACOM.Structures.PopulationSpace;
using GACOM.Structures.SegmentSpace;
using GACOM.RandomNumberGenerator.RandomGeneratorSpace;
using GACOM.RandomNumberGenerator.SamplerSpace;
using GACOM.EvolutionProcess.ElitismSpace;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.IndividualSpace;
using GACOM.EvolutionProcess.OperatorSpace.RealOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSpace.QuantumOperatorsSpace;
using GACOM.EvolutionProcess.OperatorSelectionSpace;
using GACOM.EvolutionProcess.ReproductionSpace;
using GACOM.EvolutionProcess.GenitorSelectionSpace;
using GACOM.EvolutionProcess;
using GACOM.EvaluationProcess.DecoderSpace;
using GACOM.EvaluationProcess.FitnessSpace;
```

```
using GACOM.EvaluationProcess;
using GACOM.ExecutionBlocks;
using GACOM.InitializePopulationSpace;
using GACOM.OutputSpace;
using GACOM.StopConditionSpace;
using GACOM.EvolutionProcess.RateSpace;
```

Agora, deve-se definir a classe que representa o algoritmo genético que se deseja criar. Esta classe irá, em geral, herdar a classe GA presente na biblioteca do GACOM. A definição desta classe pode ser feita da seguinte maneira:

```
public class GAF6 : GA
{
```

Dentro desta classe, deve-se definir o método construtor. A definição do método é dada abaixo:

```
public GAF6(string name)
  : base(name) { }
```

Agora, inicia-se a construção do corpo do método construtor. Em primeiro lugar, deve-se definir os geradores de números aleatórios. É possível criar novos geradores, mas o GACOM já possui um conjunto de geradores de números aleatórios. Para fazer isso, use o código mostrado abaixo:

```
IRandom[] p_random;

p_random = new IRandom[2];
p_random[0] = new LGMRandGen();
p_random[1] = new LGMRandGen();
```

Neste caso foram criados dois geradores de números aleatórios, mas o usuário pode criar quantos ele desejar. Deve-se, em seguida, definir os amostradores, determinando suas respectivas distribuições de probabilidade. Cada amostrador deve ter associado a ele um dos geradores de números aleatório criados anteriormente. Para tanto, deve-se fazer como o exposto abaixo:

```
samplers = new ISampler[2];

samplers[0] = new UniformSampler();
samplers[0].RandomGenerator = p_random[0];
samplers[1] = new UniformSampler();
samplers[1].RandomGenerator = p_random[1];
```

Neste caso foram criados dois amostradores, mas o usuário pode criar quantos ele desejar. O primeiro amostrador é sempre utilizado na inicialização das populações. Deve-se definir dois ssteady-states distindos, uma para a população clássica e outro para a quântica. O *steady-state* clássico (SteadyState) recebe como parâmetros um objeto de taxa linear e uma variável booleana. A taxa linear, por sua vez, recebe uma referência ao próprio GA, uma taxa inicial e uma final. Isto permite que o valor da taxa possa mudar ao longo da execução do GA. Já para o *steady-state* quântico, deve-se passar uma taxa referente a quantidade de

indivíduos criados a partir da população clássica e quântica. Nesse exemplo, duas instâncias do *steady-state* são utilizadas, uma para cada população:

```
SteadyState eli1 = new SteadyState(new LinearRate(this,
SteadyStateInitialRate, SteadyStateFinalRate), false);

QuantumSteadyState qSS = new QuantumSteadyState(new LinearRate(this,
SteadyStateInitialRate, SteadyStateFinalRate), new LinearRate(this,
1.0, 1.0), true);
```

O próximo passo consiste em definir a técnica de seleção de operadores. Para o GA quântico, deve-se usar um operador para cada população e o amostrador associado a ela:

```
RouletteOperatorSelection tOpSel = new RouletteOperatorSelection();
tOpSel.Sampler = samplers[1];

RouletteOperatorSelection qOpSel = new RouletteOperatorSelection();
qOpSel.Sampler = samplers[1];

RouletteOperatorSelection cOpSel = new RouletteOperatorSelection();
cOpSel.Sampler = samplers[1];

RouletteOperatorSelection opsel1 = new RouletteOperatorSelection();
opsel1.Sampler = samplers[1];

RouletteOperatorSelection opsel2 = new RouletteOperatorSelection();
opsel2.Sampler = samplers[1];
```

É possível, então, definir os operadores genéticos a serem utilizados. Neste exemplo, como os cromossomos da população de cidades são baseados em ordem, enquanto que os cromossomos da população de aviões são inteiros. Sendo assim, é preciso utilizar operadores distintos para cada uma das populações. Para cada um deles, deve-se definir um amostrador, como se segue:

```
LinearRate mutationRate = new LinearRate(this, InitialMutation,
                                                    FinalMutation);
LinearRate crossoverRate = new LinearRate(this, InitialCrossover,
                                                    FinalCrossover);

// Populacao de cidades
Operator MutOp11 = new SwapMutation(mutationRate);
MutOp11.Sampler = samplers[1];

Operator MutOp12 = new PIMutation(mutationRate);
MutOp12.Sampler = samplers[1];

Operator CroOp11 = new PMXCrossover(crossoverRate);
CroOp11.Sampler = samplers[1];

Operator CroOp12 = new OXCrossover(crossoverRate);
CroOp12.Sampler = samplers[1];

Operator CroOp13 = new CXCrossover(crossoverRate);
CroOp13.Sampler = samplers[1];
```

```
// Populacao de avioes
Operator Cro21 = new SinglePointIntegerCrossover(crossoverRate);
Cro21.Sampler = samplers[1];

Operator Cro22 = new TwoPointIntegerCrossover(crossoverRate);
Cro22.Sampler = samplers[1];

Operator Cro23 = new IntegerUniformCrossover(crossoverRate);
Cro23.Sampler = samplers[1];

Operator Mut21 = new IntegerUniformMutation(mutationRate);
Mut21.Sampler = samplers[1];
```

Em seguida, adiciona-se os operadores à técnica de seleção, especificando-se também os pesos a serem utilizados no processo de seleção dos operadores e o segmento que eles atuarão. Como neste experimento cada população possui apenas um segmento, todos os operadores serão aplicados no mesmo segmento, porém deve-se ter cuidado para atribuir os operadores corretos para as respectivas populações:

```
opsel1.addOperator(MutOp11, 1, 0);
opsel1.addOperator(MutOp12, 1, 0);
opsel1.addOperator(CroOp11, 1, 0);
opsel1.addOperator(CroOp12, 1, 0);
opsel1.addOperator(CroOp13, 1, 0);

opsel2.addOperator(Cro21, 1, 0);
opsel2.addOperator(Cro22, 1, 0);
opsel2.addOperator(Cro23, 1, 0);
opsel2.addOperator(Mut21, 1, 0);
```

O próximo trecho do código consolida uma série de definições:

```
Reproduction repr1 = new Reproduction(opsel1, eli1);
Reproduction repr2 = new Reproduction(opsel2, eli2);
GenitorSelection gensel1 = new RouletteGenitorSelection(samplers[1]);
GenitorSelection gensel2 = new RouletteGenitorSelection(samplers[1]);
Evolution evol1 = new Evolution(gensel1, repr1);
Evolution evol2 = new Evolution(gensel2, repr2);
SelectBestFromOthers selectmethod = new SelectBestFromOthers();
CoevEvaluation evaluationFunc = new CoevEvaluation();
CoevolutionaryDecoder dec = new CoevolutionaryDecoder(evaluationFunc);
CoevolutionaryEvaluation eval = new CoevolutionaryEvaluation(dec,
samplers[1], selectmethod);
InitEachIndividual indInit = new InitEachIndividual();

// População de referência
Population pop_cities = new Population(PopulationSize, "Ordem das
Cidades", eval, evol1, this, indInit);

// População de referência
Population pop_planes = new Population(PopulationSize, "Avioes", eval,
evol2, this, indInit);
ITrace trace_model = new Trace();
pop_cities.TraceModel = trace_model;
pop_planes.TraceModel = trace_model;
pop_cities.Maximization = true;
pop_planes.Maximization = true;
```

As duas primeiras linhas instanciam a classe de reprodução de cada uma população, indicando os operadores genéticos escolhidos e o *steady-state*. As duas próximas linhas definem os modelos de seleção dos genitores (no caso, por roleta), e passam o amostrador para os mesmos. Em seguida, instanciam-se as classes que controlam a evolução, passando como parâmetros a técnica de seleção e a reprodução. Então é definido o método de seleção dos indivíduos da outra população para serem usados na avaliação da população corrente. Na linha seguinte, tem-se a definição do decodificador que será utilizado pelo algoritmo genético durante a avaliação. A definição desta classe é mostrada mais a frente neste tutorial. Finalmente, define-se a instância de avaliação (passando para a mesma o decodificador utilizado, um amostrador e o método de seleção dos indivíduos da outra população). O próximo passo consiste em instanciar o modo de inicialização da população. O usuário pode criar novos tipos de inicialização, mas o GA já fornece alguns. Em seguida, cria-se a população. As populações recebem como parâmetros a quantidade de indivíduos que as constituem (tamanho da população), uma string utilizada para identificação, a instância de avaliação, a instância que define o processo de evolução, uma referência ao algoritmo genético que está sendo criado (e portanto, a própria classe que se está definindo, daí o uso da palavra reservada *this*) e a instância que define o modo de inicialização da população. Por último, instancia-se o modelo de trace, que guarda informações relativas à cada população, e defini-se o problema como de maximização para ambas as populações por se tratar de um problema de simbiose.

Após a criação da população, deve-se ainda adicionar o método de normalização linear e adicioná-la a um bloco, o que pode ser feito da seguinte maneira:

```
SimpleLinearNormalization norm = new
                        SimpleLinearNormalization(1, PopulationSize);
pop_cities.addBlock(2, norm);
pop_planes.addBlock(2, norm);
```

Finalmente, deve-se preencher as populações com os indivíduos iniciais. Os trechos de códigos abaixo mostram como isso deve ser feito, passando como informação a população e a quantidade de indivíduos que ela deve possuir:

```
private void fill_cities_population(ref Population population)
{
  Individual ind = null;
  OrderBasedSegment seg1 = null;
  OrderBasedGene gene = null;
  int TamCromo = 25; // Tamanho do cromossomo (número de cidades)
  ArrayList list;
  list = new ArrayList();

  for (int i = 0; i < TamCromo; i++)
  {
    list.Add(i);
  }

  ind = new Individual(1);
  seg1 = new OrderBasedSegment(TamCromo, list);
  for (int j = 0; j < TamCromo; j++)
  {
    gene = new OrderBasedGene();
```

```
      seg1.AddGene(gene);
    }
    ind.AddSegment(seg1);

    ind.Fitness = new RealFitness();
    ind.OriginalFitness = new RealFitness();

    population.addIndividual(ind);

    for (int j = 0; j < PopulationSize - 1; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }
```

Este código cria, inicialmente, uma instância de um indivíduo e de um segmento. Na criação do indivíduo é passado o número de segmentos que este possui. Segmentos são utilizados pelo GACOM para separar partes do cromossomo com diferentes representações. Nesse caso, como na população de cidades todo o cromossomo é baseado em ordem, o indivíduo possui apenas um segmento. Para a criação deste, é passado o números de genes que o constituirão. Após criar o segmento, adiciona-se os genes ao mesmo. Feito isso, adiciona-se o segmento ao indivíduo. As três linhas seguintes servem para indicar ao GACOM que a aptidão (*Fitness*) e a avaliação (*OriginalFitness*) são reais.

Em seguida, adiciona-se os indivíduos à população. Note que, após adicionar o primeiro indivíduo, é necessário adicionar os outros membros da população usando o método Clone. Isto garante que os indivíduos que estão sendo inseridos na população não compartilham a mesma região de memória.

Para a população de aviões utilizados a principal diferença é que o gene agora é inteiro e deve-se passar o número máximo e mínimo que cada gene pode atingir.

```
  private void fill_planes_population(ref Population population)
  {
    Individual ind = null;
    Segment seg1 = null;
    IntegerGene gene = null;
    int TamCromo = 5; // 5 avioes escolhidos
    int min = 0;
    int max = 14;

    ind = new Individual(1);
    seg1 = new Segment(TamCromo);

    for (int j = 0; j < TamCromo; j++)
    {
      gene = new IntegerGene(min,max);
      seg1.AddGene(gene);
    }

    ind.AddSegment(seg1);

    ind.Fitness = new RealFitness();
    ind.OriginalFitness = new RealFitness();
```

```
    population.addIndividual(ind);

    for (int j = 0; j < PopulationSize - 1; j++)
    {
      population.addIndividual((IIndividual)ind.Clone());
    }
  }
```

Agora, deve-se criar um vetor de populações com tamanho igual ao número de populações que devem co-evoluir e em seguida adicionar as populações recém-criadas ao algoritmo genético. Isto pode ser feito através do seguinte comando:

```
this.p_pops = new IPopulation[2];
addPopulation(pop_cities);
addPopulation(pop_planes);
```

As condições de parada do algoritmo genético são definidas através da classe *StopCondition*, informando a quantidade de gerações e de experimentos, como pode ser visto no exemplo abaixo:

```
StopCondition = new StopCondition(this, Generations, Experiments);
```

Por último, deve-se definir como os resultados serão exibidos ao usuário. O GACOM define uma classe muito simples que pode ser usada para essa finalidade. Essa classe, basicamente, mostra na tela os resultados do algoritmo genético:

```
CoevolutionaryConsoleOutput cvout = new
                                    CoevolutionaryConsoleOutput(this);
```

Além desta classe, uma outra deve ser criada para fins de avaliação. Esta nova classe deve herdar de *FitnessFuction* e deve conter o código com a avaliação dos indivíduos. A definição desta classe dever ser feita da seguinte maneira:

```
public class CoevEvaluation : FitnessFunction
{
```

O método *evaluate* da classe FitnessFuction deve ser sobrescrito nesta classe. Conforme mostramos abaixo:

```
public override Fitness evaluate(IGene[] objects,
    ref IIndividual individual, int generation)
{
```

O código completo desta classe é mostrado abaixo:

```
using GACOM.EvaluationProcess.FitnessFunctionSpace;
using GACOM.EvaluationProcess.FitnessSpace;
using GACOM.Interfaces;
using GACOM.Structures.GeneSpace;
using GACOM.Structures.SegmentSpace;
using System.Collections;
using System;
using System.IO;
```

```
namespace Coev
{
  public class CoevEvaluation : FitnessFunction
  {
    private double[] planesAuto;
    private double[,] cities;

    public CoevEvaluation()
    {
      planesAuto = new double[15];
      cities = new double[26, 2];

      //carrega os pesos dos avioes
      StreamReader sr = File.OpenText(@"D:\usr\leandro\Projects\GACOM-
Tutoriais\TSP-mod\eval\plane.txt");

      string line;
      int i = 0;

      while (((line = sr.ReadLine()) != null) && (i <
planesAuto.GetLength(0)))
      {
        line = line.Trim();
        string[] word;
        if (line.Length > 0)
        {
          string[] space = { "\t", " " };
          word = line.Split(space,
                       StringSplitOptions.RemoveEmptyEntries);

          planesAuto[i++] = Convert.ToDouble(word[0]);
        }
      }
      sr.Close();

      //carrega as localizacoes das cidades
      sr = File.OpenText(@"D:\usr\leandro\Projects\GACOM-Tutoriais\TSP-
mod\eval\city.txt");

      i = 0;

      while (((line = sr.ReadLine()) != null) && (i <
cities.GetLength(0)))
      {
        line = line.Trim();
        string[] word;
        if (line.Length > 0)
        {
          string[] space = { "\t", " " };
          word = line.Split(space,
                           StringSplitOptions.RemoveEmptyEntries);
          cities[i, 0] = Convert.ToDouble(word[0]);
          cities[i++, 1] = Convert.ToDouble(word[1]);
        }
      }
      sr.Close();
    }
```

```
    public override Fitness evaluate(IGene[] objects, ref IIndividual
individual, int generation)
    {
      try
      {
        double next; // distancia euclidiana para a proxima cidade
        double home; // distancia euclidiana para voltar para a base
        int totalCities = 0;
        double autonomia;
        int[] cityIndex = new int[25];
        int[] avioes = new int[5];

        // armazena os elemntos referentes aos avioes e as cidades
        int posa = 0;
        int posc = 0;

        for (int i = 0; i < objects.Length; i++)
        {
          if (objects[i] is IntegerGene)
          {
            avioes[posa] = (int)objects[i].Value;
            posa++;
          }
          else
          {
            cityIndex[posc] = (int)objects[posc].Value;
            posc++;
          }
        }

        ArrayList solution = new ArrayList();
        ArrayList item;
        int cont;

        for (int i = 0; i < avioes.Length; i++)
        {
          cont = 0;
          foreach (object o in solution)
          {
            if ((int)(((ArrayList)o)[0]) == avioes[i])
            {
              for (int j = 0; j < 5; j++)
              {
                ((ArrayList)o).Add(cityIndex[(i * 5) + j]);
              }
              cont = 1;
              break;
            }
          }
          if (cont == 0)
          {
            item = new ArrayList();
            item.Add(avioes[i]);
            for (int j = 0; j < 5; j++)
            {
              item.Add(cityIndex[(i * 5) + j]);
            }
            solution.Add(item);
```

```
          }
        }

        foreach (object o in solution)
        {
          autonomia = planesAuto[(int)((ArrayList)o)[0]];

          // distancia para a primeira cidade
          next = Math.Pow(cities[(int)((ArrayList)o)[1] + 1, 0] -
cities[0, 0], 2);
          next += Math.Pow(cities[(int)((ArrayList)o)[1] + 1, 1] -
cities[0, 1], 2);
          next = Math.Sqrt(next);

          home = next;

          if (autonomia >= (next + home))
          {
            totalCities++;
            autonomia -= next;
          }
          else continue;

          for (int i = 1; i < ((ArrayList)o).Count - 1; i++)
          {
            // distancia da proxima cidade
            next = Math.Pow((cities[(int)((ArrayList)o)[i] + 1, 0] -
cities[(int)((ArrayList)o)[i + 1] + 1, 0]), 2);
            next += Math.Pow((cities[(int)((ArrayList)o)[i] + 1, 1] -
cities[(int)((ArrayList)o)[i + 1] + 1, 1]), 2);
            next =  Math.Sqrt(next);

            // distancia para voltar da proxima cidade para base
            home = Math.Pow((cities[(int)((ArrayList)o)[i + 1] + 1, 0]
- cities[0, 0]), 2);
            home += Math.Pow((cities[(int)((ArrayList)o)[i + 1] + 1, 1]
- cities[0, 1]), 2);
            home =  Math.Sqrt(home);

    // verifica se tem combustivel para ir pra proxima cidade e voltar

            if (autonomia >= (next + home))
            {
              totalCities++;
              autonomia -= next;
            }
            else break;
          }
        }

        individual.OriginalFitness = new
                        RealFitness(Convert.ToDouble(totalCities));
        individual.Fitness = new
                        RealFitness(Convert.ToDouble(totalCities));

        return individual.Fitness;
      }
      catch (Exception e)
      {
```

```
            throw new Exception("CoevEvaluation: (evaluate) " + e.Message);
          }
        }
      }
    }
```

Para a avaliação do co-evolucionáro, é preciso tratar os objetos de entrada uma vez que esses intercalam de acordo com a população que está sendo avaliada, para isso é feita a checagem de se o gene é inteiro ou real, podendo saber qual a população a qual aquele gene pertence. Após a execução da avaliação (que no caso é o número de cidades percorridas). deve-se definir o *fitness* do indivíduo como o resultado desse método. O *OriginalFitness* continua sendo a avaliação original de cada objetivo e o *Fitness* consiste da aptidão que será normalizada para ser utilizada na seleção do indivíduo.

Por fim, precisamos definir o programa principal que irá chamar o GAcom. A seguir temos um exemplo de programa que chama o GACOM:

```
using System;

namespace Coev
{
  class Program
  {
    static void Main(string[] args)
    {
      CoevGA ga = new CoevGA("TSPGA", null);

      ga.Generations = 100;
      ga.PopulationSize = 60;
      ga.InitialCrossover = 0.85;
      ga.FinalCrossover = 0.45;
      ga.InitialMutation = 0.08;
      ga.FinalMutation = 0.8;
      ga.InitialSteadyState = 0.6;
      ga.FinalSteadyState = 0.4;
      ga.Experiments = 1;

      ga.exec();

      Console.ReadKey();
    }
  }
}
```