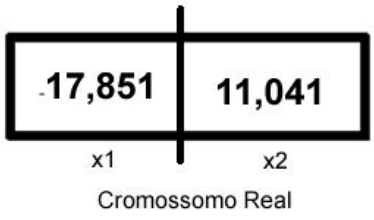
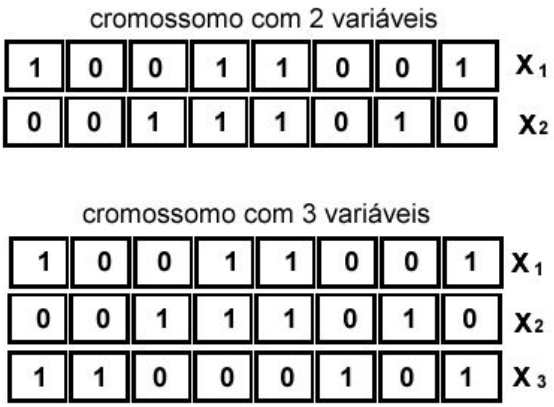
Atividade 2

1. Introdução

A presente atividade ilustra o uso de representação binária para solucionar o problema de minimização da função *Camel Back*. A





Há duas importantes questões na representação de números reais em binários:

1. Intervalo de domínio de cada uma das variáveis.
2. Precisão desejada.

Assim, ao utilizar a representação binária no espaço dos genótipos para realizar a busca por soluções no domínio dos reais no espaço dos fenótipos, precisamos considerar uma decodificação da representação binária que garanta a precisão desejada e esteja dentro do domínio das variáveis.

Exemplo - Decodificação

$$f(x) = x\,\mathrm{sen}(10\pi x) + 1 \qquad -1 \le x \le 2\,.$$

$$s_1 = 10001011101101010001111 \qquad b_{10} = (10001011101101010001111)_2 = 2288967$$

$$x = \min + (\max - \min)\frac{b_{10}}{2^l - 1} \qquad x_1 = -1 + (2+1)\frac{2.288.967}{\left(2^{22} - 1\right)} = 0{,}637197$$

2. **Implementação no ProOF**

Passo 1: Criar pacote **ProOF.apl.sample1.problem.CBbin** ilustrado na Figura 1.
Dica: Selecione ProOF.apl.sample1.problem, clique o botão direito do mouse e selecione as opções New e Java Package. Assim o pacote será criado em **ProOF.apl.sample1.problem**

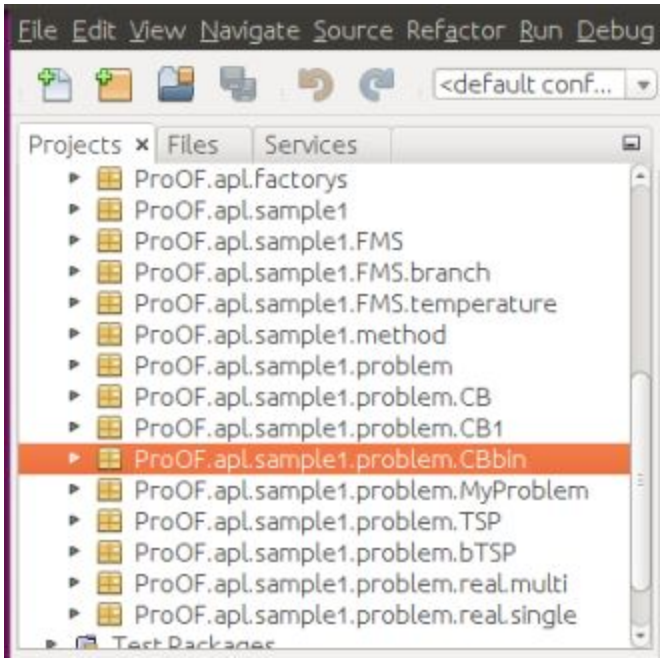


Figura 1. Criando pacote CBbin

Passo 2: Copiar os arquivos que estão no pacote **ProOF.apl.sample1.problem.CB** para **ProOF.apl.sample1.problem.CBbin**, **exceto o arquivo CBInstance.java**. Os arquivos devem ser renomeados para **CBbin** como ilustrado na Figura 2.
Dica: Utilize a opção Refactor ao colar os arquivos copiados para o novo pacote CBbin
Dica: Para renomear, selecione o arquivo e utilize as teclas de atalho CTR+R.

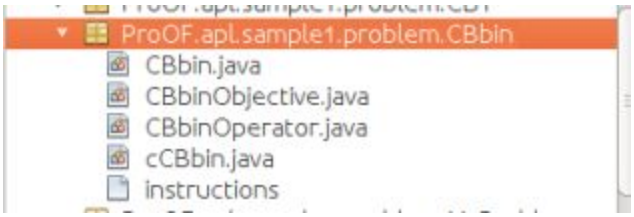


Figura 2. Arquivos copiados e renomeados.

Passo 3: Abrir o arquivo **cCBbin.java** e incluir as alterações abaixo como ilustrado na Figura 3.

1. Declarar as estruturas abaixo para armazenar as codificações:

a. Matriz **coordBin** para armazenar a codificação binária (matriz de inteiros).
b. Vetor **coordReal** para armazenar a codificação real (vetor double).

2. Criar as estruturas considerando as dimensões dim (ou dimReal como utilizado em sala de aula) e dimBin.
3. Implementar o método **copy** que permite replicar a codificação binária armazenada em coordBin.

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package ProOF.apl.sample1.problem.CBbin;
6  import ProOF.opt.abst.problem.meta.codification.Codification;
7
8  /**...4 lines */
12 public class cCBbin extends Codification<CBbin, cCBbin> {
13     protected int coordBin[][];
14     protected double coordReal[];
15
16     public cCBbin(CBbin prob) {
17         this.coordBin = new int [prob.inst.dim][prob.inst.dimBin];
18         this.coordReal = new double[prob.inst.dim];
19     }
20     @Override
22     public void copy(CBbin prob, cCBbin source) throws Exception {
23         //System.arraycopy(source.path, 0, this.path, 0, this.path.length);
24         for(int i=0; i< prob.inst.dim;i++)
25         for(int j=0; j< prob.inst.dimBin;j++){
26             this.coordBin[i][j]=source.coordBin[i][j];
27         }
28     }
29     @Override
30     public cCBbin build(CBbin prob) throws Exception {
           return new cCBbin(prob);
```

1
2
3

Figura 3. Alterações no arquivo cCBbin.java

Passo 4: Abrir o arquivo **CBbinObjective.java** e executar as seguintes alterações:

1. No método **evaluate**, incluir uma chamada ao método **dCBBin(prob,codif)** que implementa a decodificação da representação binária (espaço dos genótipos) para a representação real (espaço dos fenótipos). Figura 4.1 ilustra como incluir tal chamada no código. Observe que o cálculo do fitness continuará sendo feito utilizando a representação real que agora é obtida pela decodificação da representação binária.
2. O método **dCBBin()** deverá ser implementado como ilustrado na Figura 4.2. O processo de decodificação será explicado através de um exemplo. Suponha que estejamos utilizando uma cadeia binária de tamanho 22 (dimBin=22) que será convertida para um valor real no intervalo -1□x□2. O primeiro passo é achar o valor na base 10 (b_{10}) para a representação binária considerada (s_1) como ilustrado a seguir.

$$s_1 = 1000101110110101000111$$

$$b_{10} = (10001011101101010001111)_2 = 2288967$$

As **linhas 34 a 38** na Figura 4.2 realizam essa conversão, onde s1 está armazenada em coordBin e b_{10} é armazenado na variável **value** declarada como sendo do tipo inteiro longo na **linha 35**. Por último, o valor em b_{10} precisa ser mapeado para o intervalo considerado no domínio dos reais como ilustrado a seguir.

$$x = \min + (\max - \min)\frac{b_{10}}{2^l - 1} \qquad x_1 = -1 + (2+1)\frac{2.288.967}{(2^{22}-1)} = 0{,}637197$$

As **linhas 39 a 41** realizam tal mapeamento considerando os valores min e max fornecidos. Observe que codif.coordReal[i] armazena o valor em reais da cadeia binária na linha i da matriz coordBin[i][ ].

```
Source  History
1  ...4 lines
5  package ProOF.apl.sample1.problem.CBbin;
6
7  import ProOF.opt.abst.problem.meta.objective.SingleObjective;
8
9  /**...4 lines */
13 public class CBbinObjective extends SingleObjective<CBbin, cCBbin, CBbinObjective> {
14     public CBbinObjective() throws Exception {
15         super();
16     }
17     @Override
       public void evaluate(CBbin prob, cCBbin codif) throws Exception {
           double fitness = 0;
20         this.dCBBin(prob, codif);   1
21         fitness = prob.inst.Coef[0]*Math.pow(codif.coordReal[0],2.0) +
22                 prob.inst.Coef[1]*Math.pow(codif.coordReal[0],4) +
23                 prob.inst.Coef[2]*Math.pow(codif.coordReal[0],6) +
24                 prob.inst.Coef[3]*codif.coordReal[0]*codif.coordReal[1] +
25                 prob.inst.Coef[4]*Math.pow(codif.coordReal[1],2);
26         set(fitness);       //set de fitness to the ProOF
27     }
28     @Override
       public CBbinObjective build(CBbin prob) throws Exception {
30         return new CBbinObjective();
31     }
32     private void dCBBin(CBbin prob, cCBbin codif){
```

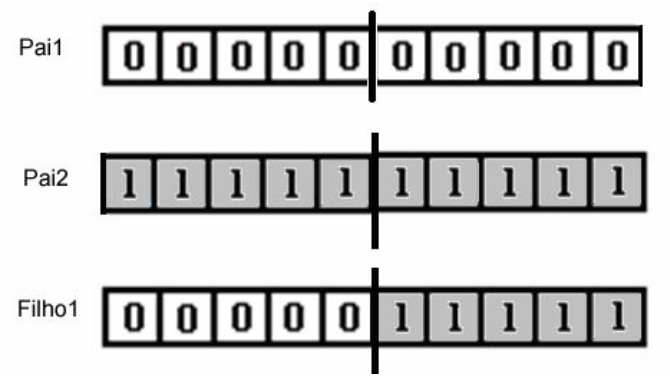
Figura 4.1. Inserção de chamada para função de decodificação

```
22                  prob.inst.Coef[1]*Math.pow(codif.coordReal[0],4) +
23                  prob.inst.Coef[2]*Math.pow(codif.coordReal[0],6) +
24                  prob.inst.Coef[3]*codif.coordReal[0]*codif.coordReal[1] +
25                  prob.inst.Coef[4]*Math.pow(codif.coordReal[1],2);
26              set(fitness);       //set de fitness to the ProOF
27          }
28          @Override
            public CBbinObjective build(CBbin prob) throws Exception {
30              return new CBbinObjective();
31          }
32          private void dCBBin(CBbin prob, cCBbin codif){
33
34              for(int i=0; i<prob.inst.dim;i++){
35              long value = 0;
36              for(int j=0; j<prob.inst.dimBin;j++){
37                  value += Math.pow(2, prob.inst.dimBin-j)*codif.coordBin[i][j];
38              }
39              double aux;
40                  aux = value/(Math.pow(2,prob.inst.dimBin)-1);
41              codif.coordReal[i]= (double) ( prob.inst.min+ (prob.inst.max-prob.inst.min)*aux);
42             // System.out.println("***"+value+"**"+aux+"**"+codif.coordReal[i]);
43              }
44          }
45      }
46
```

Figura 4.2. Implementação da função de decodificação.

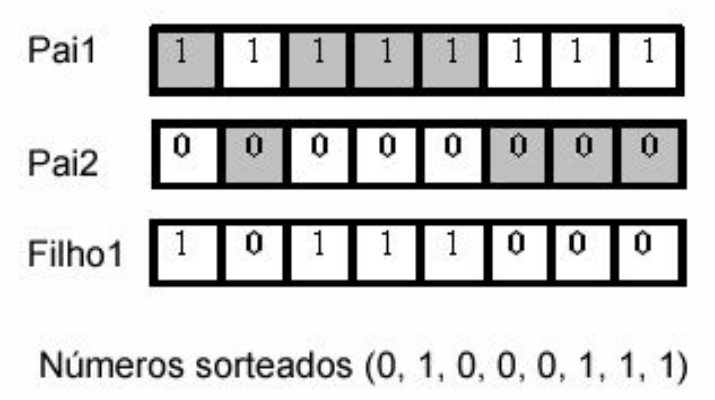
Passo 5: Abrir o arquivo **CBbinOperator.java** e executar as seguintes alterações:

1. Alterar o método **name** como ilustrado na Figura 5.1
2. Alterar o método **build** para incluir os operadores ilustrados na Figura 5.1
3. Na classe **RandomInit**
    a. Alterar o método **name** como ilustrado na Figura 5.2. Essa classe implementa o operador de inicialização aleatória da representação binária.
    b. Alterar o método **initialize** como ilustrado na Figura 5.2. Observe que há 50% de chance de um valor 0 ou 1 ser atribuído à cadeia binária.
4. Na classe RandomMut
    a. Alterar o método **name** como ilustrado na Figura 5.3.
    b. Alterar o método **mutation** como ilustrado na Figura 5.3. O método implementa o operador de mutação que executa basicamente três passos. Primeiro, seleciona na linha 64 aleatoriamente uma ordenada para ser alterada das dim (dimReal) possíveis. Em seguida, seleciona aleatoriamente na linha 63 uma posição da cadeia binária. Por último, o valor binário da posição selecionada é invertido na linha 65.
5. Na classe onePointCross
    a. Alterar o método **name** como ilustrado na Figura 5.4.
    b. Alterar o método **crossover** como ilustrado na Figura 5.4. O método implementa o operador one-point-crossover que executa a recombinação de dois indivíduos como ilustrado a seguir.



Na **linha 77**, um novo indivíduo **child** é declarado. Em seguida, um ponto de corte é aleatoriamente selecionado entre as dimBin posições possíveis. Nas **linhas 79 a 87**, todas dim (dimReal) cadeias binárias são recombinadas e armazenadas em child considerando o ponto de corte.

6. Na classe uniformCross
    a. Alterar o método **name** como ilustrado na Figura 5.5.
    b. Alterar o método **crossover** como ilustrado na Figura 5.5. O método implementa o operador uniform-crossover que executa a recombinação de dois indivíduos como ilustrado a seguir.



Na **linha 99**, um novo indivíduo child é declarado. Nas **linhas 100 a 109**, todas as dim (dimReal) cadeias binárias são recombinadas e armazenadas em child considerando 50% de chance do valor binário armazenado em child ser recebido do pai 1 ou do pai 2.

Figura 5.1. Declarando os operadores

Figura 5.2. Implementando a inicialização aleatória

```
57      private class RandomMut extends oMutation<CBbin, cCBbin>{
58          @Override
59          public String name() {
60              return "Random Mutation";
61          }
62          @Override
63          public void mutation(CBbin prob, cCBbin ind) throws Exception {
64              int rnd1 = (int) (Math.random() * prob.inst.dim);
65              int rnd2 = (int) (Math.random() * prob.inst.dimBin);
66              ind.coordBin[rnd1][rnd2] = 1-ind.coordBin[rnd1][rnd2];
67          }
68
69      }
```

5

6

Figura 5.3. Implementando a mutação aleatória.

```
70      private class onePointCross extends oCrossover<CBbin, cCBbin>{
71          @Override
72          public String name() {
73              return "onePointCrossover";
74          }
75          @Override
76          public cCBbin crossover(CBbin prob, cCBbin ind1, cCBbin ind2) throws Exception {
77              cCBbin child = ind1.build(prob);
78              int cut = (int) (Math.random() * prob.inst.dimBin);
79              for(int i=0; i< prob.inst.dim; i++){
80                  for(int j=0; j<cut;j++){
81                    child.coordBin[i][j]= ind1.coordBin[i][j];
82                  }
83                  for(int j=cut; j<prob.inst.dimBin;j++){
84                    child.coordBin[i][j]= ind2.coordBin[i][j];
85                  }
86
87              }
88              return child;
89          }
90      }
```

7

8

Figura 5.4. Implementando one-point-crossover.

```
92      private class uniformCross extends oCrossover<CBbin, cCBbin>{
93          @Override
94          public String name() {
95              return "uniformCrossover";
96          }
97          @Override
98          public cCBbin crossover(CBbin prob, cCBbin ind1, cCBbin ind2) throws Exception {
99              cCBbin child = ind1.build(prob);
100             for(int i=0; i< prob.inst.dim; i++){
101                 for(int j=0; j<prob.inst.dimBin;j++){
102                     if(Math.random()<0.5){
103                        child.coordBin[i][j]= ind1.coordBin[i][j];
104                     }
105                     else{
106                        child.coordBin[i][j]= ind2.coordBin[i][j];
107                     }
108                 }
109             }
110             return child;
111         }
112     }
```

9

10

Figura 5.5. Implementando uniform- crossover.

Passo 6: Abrir o arquivo **CBbin.java** e executar as seguintes alterações:

1. Na verdade não se trata de uma alteração, apenas destacamos que o objeto **inst** pertence a classe **CBInstance**. Isso significa que não precisamos criar uma classe **CBbinInstance** já que podemos utilizar o mesmo arquivo de instâncias do problema **CB com representação real** implementado anteriormente.
2. Alterar o método **name** como ilustrado na Figura 6.

```
14  /**...4 lines */
18  public class CBbin extends Problem<BestSol>{
19      public final CBInstance inst = new CBInstance();   1
20
21
22      @Override
24      public String name() {
25          return "CBbin";   2
26      }
```

Figura 6. Alterando CBbin.java

Passo 7: Abrir o arquivo **fProblem.java** no pacote **ProOF.apl.factorys** e executar as seguintes alterações:

1. Importar o pacote **CBbin** como ilustrado na Figura 7.1
2. Incluir o problema **CBbin** como ilustrado na Figura 7.2

```
10  import ProOF.apl.sample1.problem.CB1.CB1;
11  import ProOF.apl.sample1.problem.CB.CB;
12  import ProOF.apl.sample1.problem.CBbin.CBbin;   1
13  import ProOF.apl.sample1.problem.MyProblem.MP;
14  import ProOF.apl.sample1.problem.TSP.TSP;
15  import ProOF.com.language.Factory;
16  import ProOF.gen.codification.FunctionSingle.SingleBinRealFunction;
17  import ProOF.gen.codification.FunctionSingle.SingleRealFunction;
18  import ProOF.gen.codification.FunctionMulti.MultiBinRealFunction;
19  import ProOF.gen.codification.FunctionMulti.MultiRealFunction;
20  import ProOF.opt.abst.problem.meta.Problem;
21
22  /**...4 lines */
26  public final class fProblem extends Factory<Problem>{
27      public static final fProblem obj = new fProblem();
28
29      @Override
31      public String name() {
32          return "Problem";
33      }
```

Figura 7.1. Importando CBbin

```
34      @Override
        public Problem build(int index) throws Exception {
36          switch(index){
37              case 0: return new TSP();
38              case 1: return new SingleRealFunction   (fRealSingle.obj,   fRealOperator.obj)
39              case 2: return new SingleBinRealFunction(fRealSingle.obj,   fBinRealOperator.ob
40              case 3: return new MultiRealFunction    (fRealMulti.obj,    fRealOperator.obj)
41              case 4: return new MultiBinRealFunction (fRealMulti.obj,    fBinRealOperator.ob
42              case 5: return new MP();
43              case 6: return new CB();
44              case 7: return new CBbin();   2
45              case 8: return new CB1();
46          }
47          return null;
48      }
49  }
50
```

Figura 7.2. Incluindo o problema CBBin

3.Conclusão

A implementação da codificação binária para a função Camel Back permitirá a realização de uma atividade (Atividade 3) na aula do dia 28/04. Por isso, torna-se imprescindível que os alunos executem a presente Atividade 2 até a próxima aula.