

ProOF

Professional Optimization Framework: módulo **Java**

Tutorial Versão 2.2

Márcio da Silva Arantes

Conteúdo

1. Introdução.....	4
2. <i>Download</i> , Instalação e Teste do Ambiente.....	4
2.1. Instalação do Ambiente.....	6
2.2. Passos extras para usuários do Cplex	6
2.3. Fazendo um teste do ambiente.....	7
2.4. Preparação para desenvolver novos códigos no ProOF	11
3. Método Mono-Objetivo: <i>Genetic Algorithm</i>	14
4. Especializando o Caixeiro Viajante para o GA	21
5. Função Multimodal Mono-Objetivo	32
6. Modelando o TSP com o uso do Cplex no ProOF	33
7. Especializando o TSP ser resolvido pelo RFFO	33
8. Introdução aos aspectos avançados do ProOF	33
9. Implementando o método exato Branch&Bound	34
10. Especializando o TSP para o Branch&Bound.....	34
11. Implementando uma heurística gulosa genérica	34
12. Implementando o método NSGA-II para problemas multiobjetivo	34
13. Definindo uma função multiobjetivo	34
14. Conclusões	34

Resumo

Neste documento é apresentado o tutorial do ProOF (*Professional Optimization Framework*). Essa ferramenta computacional é voltada a auxiliar profissionais da área de otimização nas suas implementações de métodos/problemas e a realização de testes computacionais. O ProOF é dividido em 3 módulos chamados: **Client**, **Java** e **Cpp** cada um tendo um tutorial específico de usabilidade. Aqui a ferramenta é apresentada o módulo **Java** do ponto de vista de sua usabilidade visando ensinar ao usuário como desenvolver códigos em Java para o ProOF, sendo detalhado apenas o básico dos demais módulos quando necessário.

1. Introdução

Neste tutorial serão apresentados os passos necessários para incluir no ProOF métodos e problemas com características variadas. Todos os métodos e problemas aqui descritos já se encontram junto com o código baixado. Os passos para a implementação dos métodos e problemas ilustrados neste tutorial foram definidos com o uso da linguagem Java.

A seção 2 descreve como ter acesso a ProOF e realizar sua instalação. Na seção 3, é apresentado como incluir um *genetic algorithm* (GA) no ambiente e na seção 4 são definidos os passos necessários para incluir o problema TSP e para especializá-lo para o GA. A seção 5 propõe o uso de uma estrutura chamada FMS para facilitar os passos quando se deseja incluir problemas usando apenas variáveis reais. Isso é ilustrado utilizando a função ACK. Na seção 6 é apresentado como modelar o TSP para ser resolvido no ProOF com o uso do solver IBM Ilog Cplex. Na seção 7 é apresentado como especializar o modelo TSP desenvolvido anteriormente para ser resolvido pelo método Relax&Fix combinado com Fix&Optimize inserido no ambiente com o nome de RFFO.

Na seção 8 em diante tratará de introduzir de aspectos mais avançados do ProOF. Como a criação de novas estruturas FMS para a modelagem de métodos mais complexos, fazer modelos combinados com o uso de metaheurísticas, criação de novos operadores e tratamento de problemas multi-objetivo. Em seguida, na seção 9 uma FMS será desenvolvida para incluir no ProOF o método exato *Branch & Bound*. Na seção 10 é ilustrado como especializar o TSP para ser resolvido pelo *Branch & Bound*, na seção 11 é proposta uma simples heurística gulosa para resolver o problema TSP reaproveitando a estrutura FMS desenvolvida para o *Branch & Bound*. Para demonstrar que o ProOF pode trabalhar com problemas multi-objetivo, o algoritmo NSGA-II e uma função multi-objetivo são implementados nas seções 12 e 13, respectivamente. Ao fim a seção 14 conclui este tutorial.

2. Download, Instalação e Teste do Ambiente

Antes de iniciar a implementação dos passos para incluir métodos e problemas no ProOF, o usuário precisará baixar e instalar o ambiente em seu computador. A última versão do ProOF está disponível em <http://lcrserver.icmc.usp.br/projects/release/wiki/>. Na página *web* estão os detalhes e requisitos de sistema para instalação e também disponíveis tutoriais específicos para programar no ProOF usando as linguagem C++ e Java. Uma vez que o site do ProOF poderá sofrer

atualizações futuras, este texto não definirá os detalhes de como realizar o *download* do ambiente. Logo as instruções de *download* do ambiente devem obtidas diretamente do site. Abaixo seguem apenas os passos para a instalação da versão 2.1 do ProOF e as informações relevantes para tornar este texto auto contido.

Assim baixe a versão 2.1 completa do ProOF no site, deve ser em um arquivo chamado **ProOF2.1.zip**, descompacte o arquivo em um diretório qualquer de seu computador, exemplo (C:/ProOF2.1/)

Dica 2.1: erro de caminho

*É necessário que o caminho não tenha espaços em branco, pois caminhos com espaços em branco não são aceitos pelo módulo **Client**. Assim um caminho como (C:/Usuários/Jose/Meus Documentos/ProOF2.1/) não será aceito.*

Após descompactado o arquivo, entre no diretório que deverá conter o conteúdo ilustrado na Figura 1. Nesse diretório haverá um subdiretório `Instances` contendo arquivos de instâncias para os problemas já adicionados como exemplos no ProOF. Um subdiretório `Languages` programas capazes de compilar o código do usuário nas linguagens C++ e Java. O subdiretório **ProOFJava** é um projeto do NetBeans e do qual o usuário desenvolverá os códigos tratados nesse tutorial. Os arquivos `exec.bat` e `exec.sh` são scripts para terminais do windows e linux respectivamente para executar o módulo **Client** do ProOF. E por fim o arquivo `ProOFClient.jar` é executável do módulo **Client**.







Nome	Data de modificaç...	Tipo	Tamanho
 Instances	19/03/2015 11:49	Pasta de arquivos	
 languages	18/03/2015 12:53	Pasta de arquivos	
 ProOFJava	19/03/2015 11:47	Pasta de arquivos	
 exec.bat	03/11/2014 19:27	Arquivo em Lotes ...	1 KB
 exec.sh	03/11/2014 19:50	Shell Script	1 KB
 ProOFClient.jar	19/03/2015 11:44	Arquivo JAR	1.074 KB

Figura 1: Diretório do ProOF 2.1.

2.1. Instalação do Ambiente

Visto isso, antes de iniciar a execução ou adicionar novos códigos para o ProOF primeiro deverá ser feita a configuração do ambiente para o seu computador, a configuração precisará ser feita somente uma vez. O primeiro passo é ter instalado o Java SE Development Kit 8 e a IDE NetBeans 8.0.1, caso não tenha ainda baixe-os nos respectivos links:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html?ssSourceSiteId=otnpt>

<https://netbeans.org/>

O segundo passo é configurar o compilador Java nas variáveis de ambiente do seu sistema operacional. Esse passo costuma ser feito automaticamente durante a instalação do Java, assim abra o terminal e verifique se o comando `java` existe. Caso seja escrito comando não encontrado veja nas instruções do Java como configurá-lo nas variáveis de ambiente do seu sistema operacional.

2.2. Passos extras para usuários do Cplex

Esse terceiro passo é apenas para os usuários que vai utilizar o ProOF em conjunto com o *solver* IBM Ilog Cplex. Assim primeiramente baixe e instale o Cplex em seu sistema, depois verifique pelo terminal se o Cplex está configurado nas variáveis de ambiente, verifique se o comando (`cplex`) existe. Ao fim, copie o arquivo `cplex.jar` contido nos diretórios de instalação do Cplex para dentro do diretório das bibliotecas do projeto **ProOFJava**, substituído o arquivo atual, exemplo:

Copie `C:/.../IBM/ILOG/.../cplex/lib/cplex.jar`

Para `C:/ProOF2.1/ProOFJava/lib/cplex.jar`

Ainda no terceiro passo é preciso modificar o arquivo, `Java.cfg` que está dentro do subdiretório `Languages` do ProOF. Assim indique onde está o diretório com os arquivos de execução do Cplex instalado, substitua o caminho na 2ª linha conforme exemplo abaixo.

Atual `"H:/Programs/IBM/ILOG/CPLEX_Studio125/cplex/bin/x64_win64/"`

Depois `"C:/.../IBM/ILOG/.../cplex/bin/.../"`

2.3. Fazendo um teste do ambiente.

Caso a instalação tenha ocorrido bem um teste preliminar do ambiente já poderá ser feita. Para testar se a configuração do ambiente está correta será executado um método e um problema já desenvolvidos dentro do ProOF. Assim, inicie executando o módulo **Client**¹.

windows: duplo click no arquivo `exec.bat`

linux: execute o arquivo `exec.sh` pelo terminal

Após abrir a janela de execução do módulo **Client**, siga os 6 passos apresentados na Figura 2, para compilar e catalogar os códigos de exemplo do projeto **ProOFJava**. Será nesse projeto também que terá os futuros códigos desenvolvidos pelo usuário.

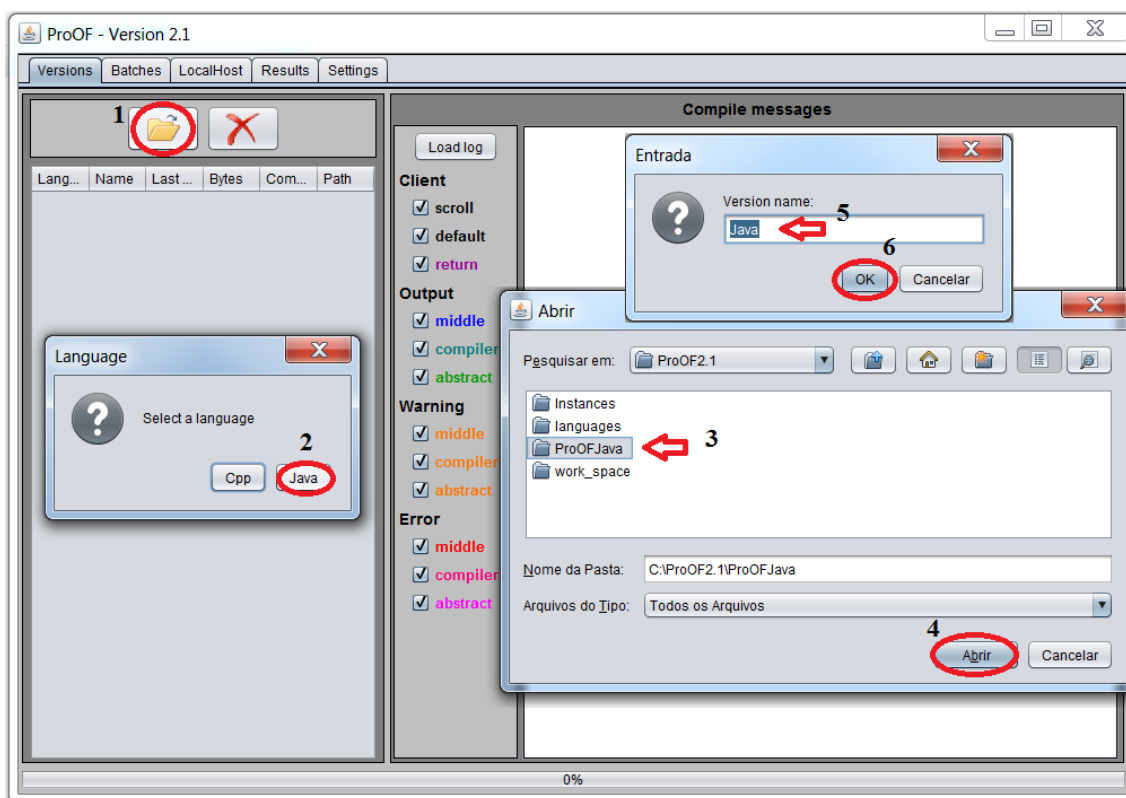


Figura 2: Preparando para compilando e catalogar o projeto ProOFJava.

A Figura 3 apresenta como deverá apresentar-se a tela após o código ser compilado e catalogado, caso o código tenha sido compilado e catalogado com sucesso deverá aparecer a mensagem indicada em 1 na figura e a versão será apresentada na tabela a esquerda conforme indicado em 2. Se qualquer erro ocorrer deverá ser apresentado nas mensagens de compilação a direita e nas cores indicadas em 3 para cada nível de erro. Os erros mais comuns são erros de

¹ pode demorar alguns segundos par abrir na primeira vez que executar

compilação (**compiler**), esses ocorrem quando o usuário comete algum erro de sintaxe nos códigos desenvolvidos. Erros de abstração (**abstract**), ocorrem quando o usuário errar ao seguir os passos do ProOF no desenvolvimento de métodos de problemas para o ambiente.

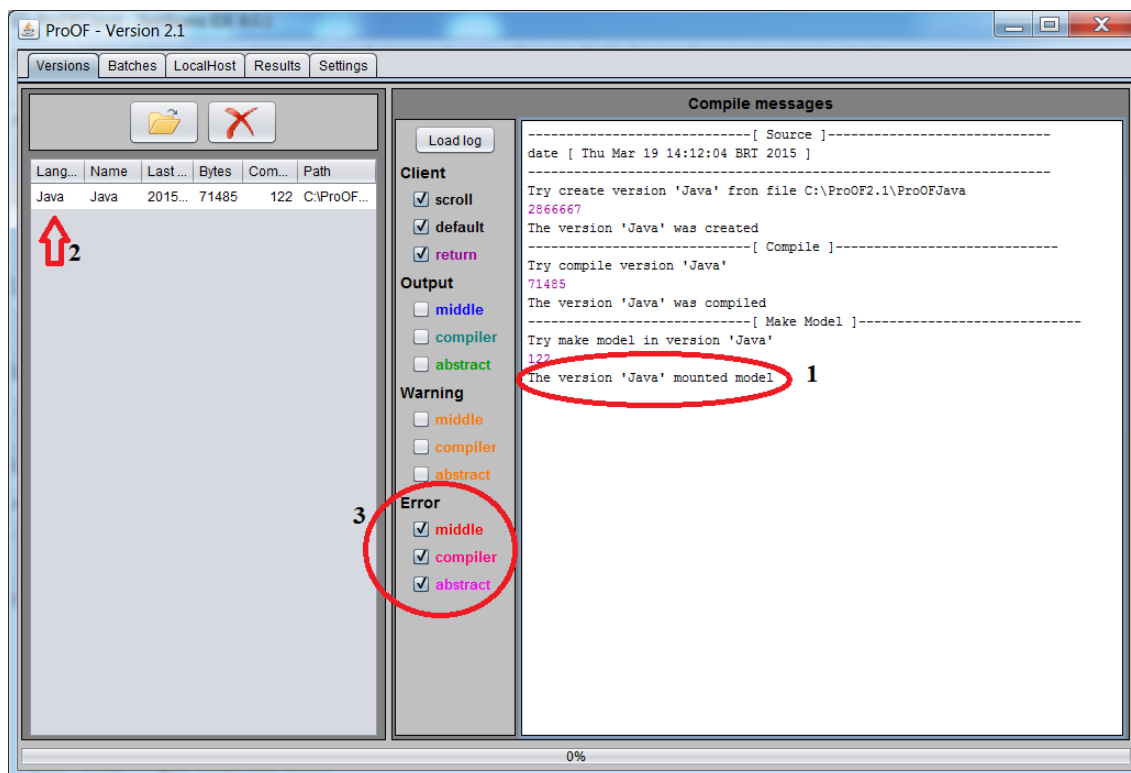


Figura 3: Compilando e catalogando o projeto ProOFJava.

Após catalogado o código uma tarefa pode ser montada na aba **Batches**, assim estabeleça uma execução do método GA para resolver o problema TSP. Seguindo os passos conforme apresentado na Figura 4 serão montadas 4 tarefas para execução. Cada execução do GA será aplicada sobre cada uma das instâncias selecionadas nos passos de 6-9. Ao fim, o passo 10 envia as tarefas para serem preparadas para a execução, a tela passará para a aba **LocalHost** conforme apresentada na Figura 5. Assim, na Figura 5, os passos de 1-3 enviam as 4 tarefas estabelecidas para a execução, onde o passo 3 apresenta a tela de visualização da execução em andamento, Figura 6.

Ao terminar todas as execuções os resultados podem ser analisados. Para isso, o usuário deve estar na tela **Results** e seguir os 4 passos estabelecidos na Figura 7. Os resultados podem ser abertos em aplicativos de planilhas para visualização. É esperado que nenhum problema ocorra durante esse teste preliminar. Esses passos são as linhas gerais para uso do módulo **Client** que é responsável por manter uma interface gráfica do ProOF e executar os códigos do usuário independentemente da linguagem utilizada: seja C++ ou Java.

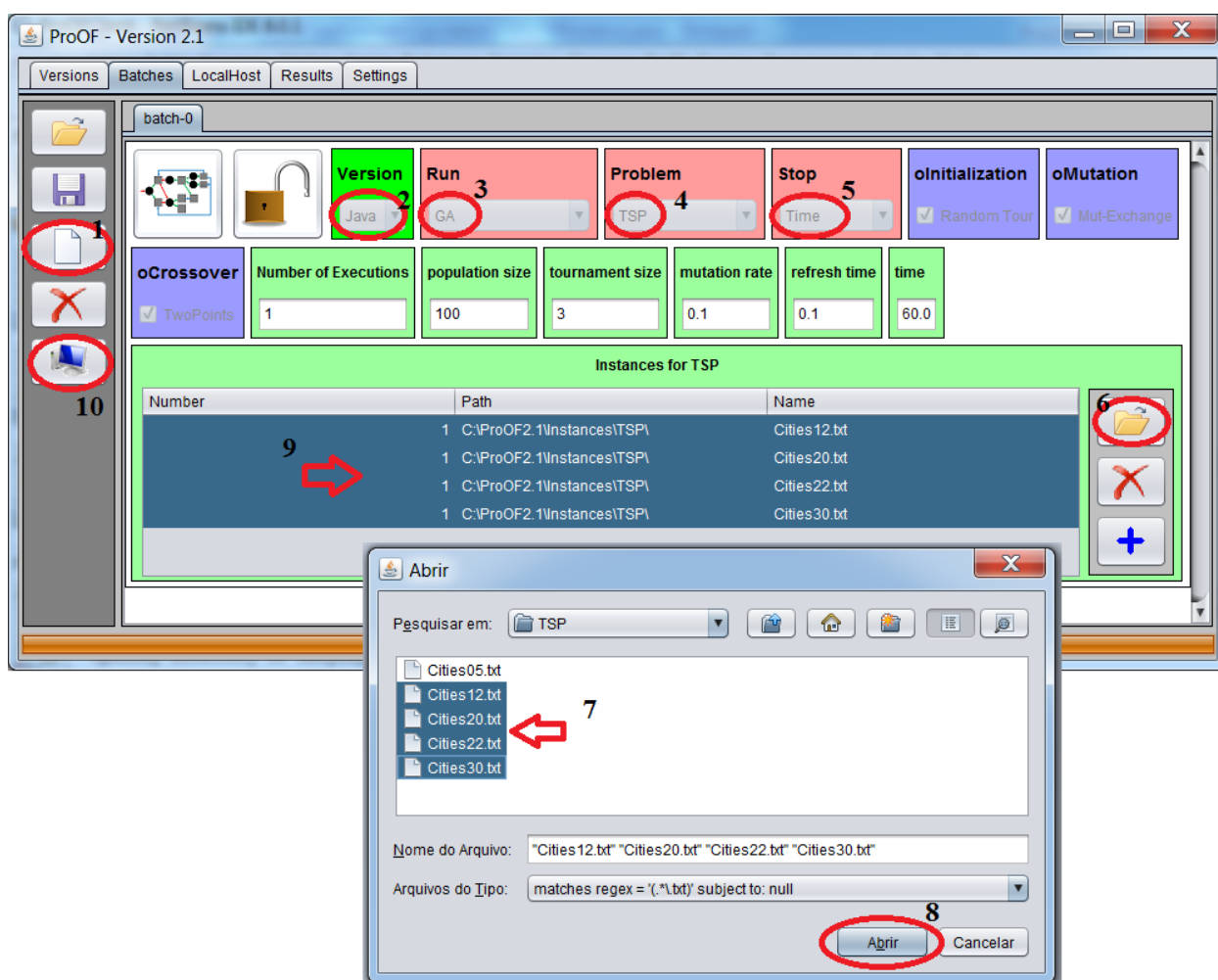


Figura 4: Montando um conjunto de tarefas para execução.

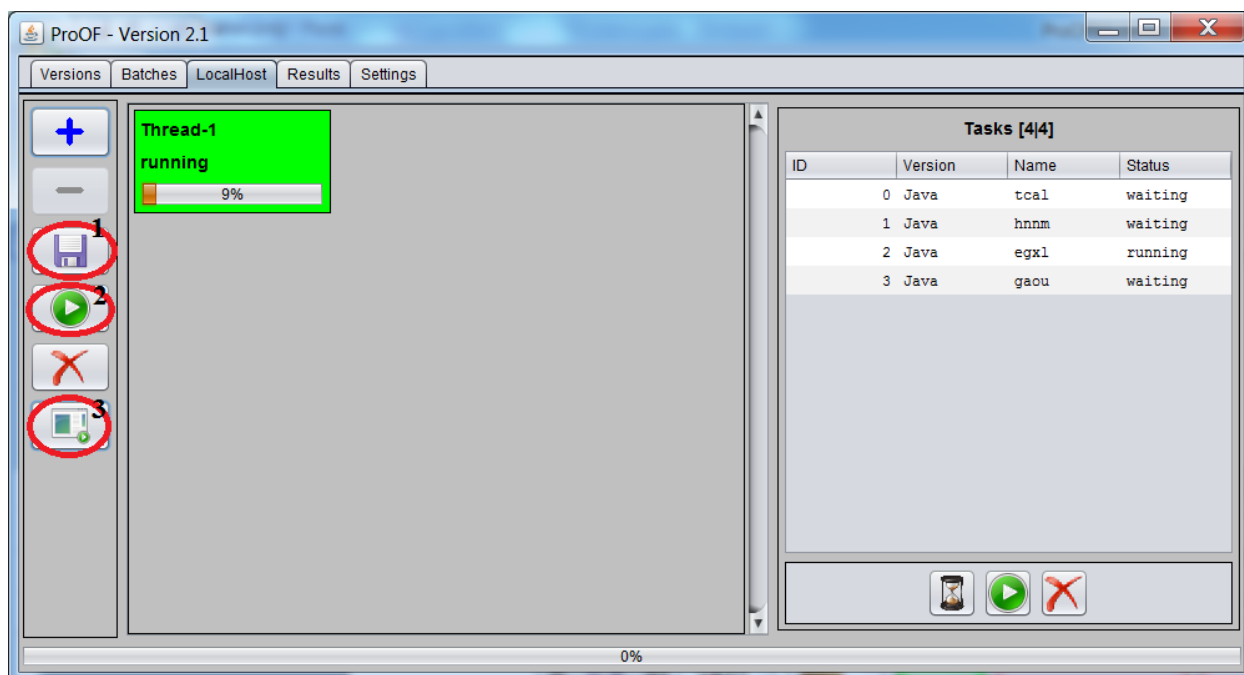


Figura 5: Executando as tarefas.

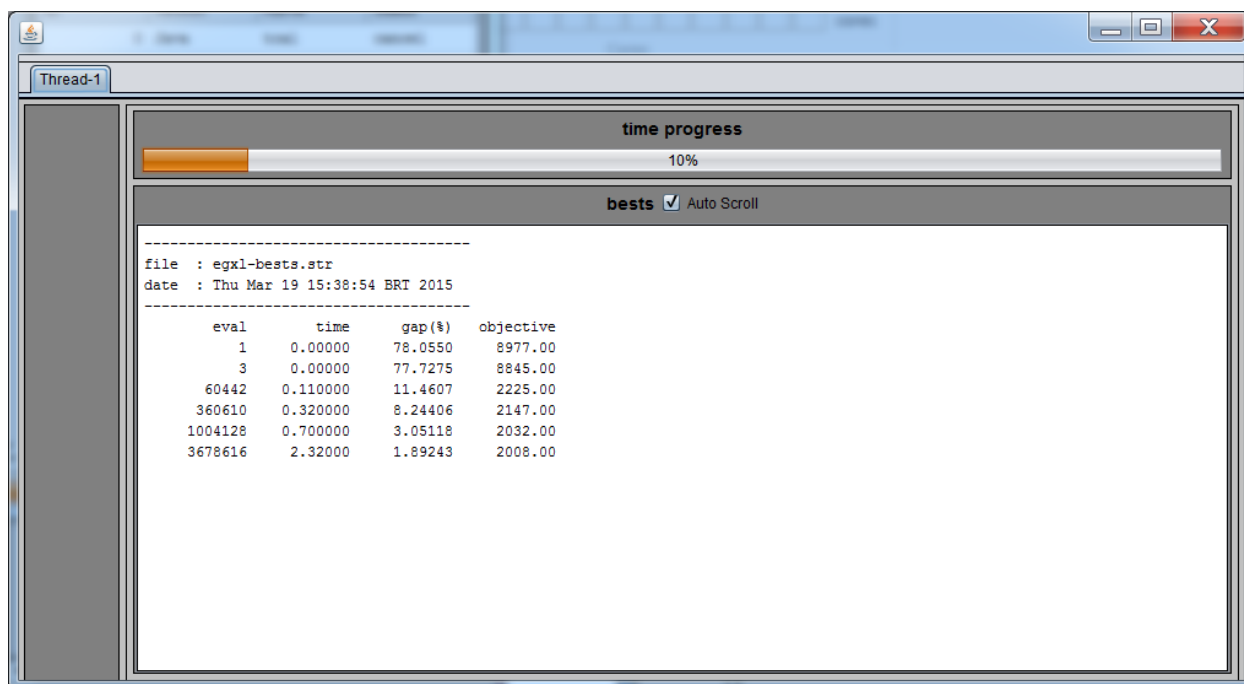


Figura 6: Visualizando a execução em andamento.

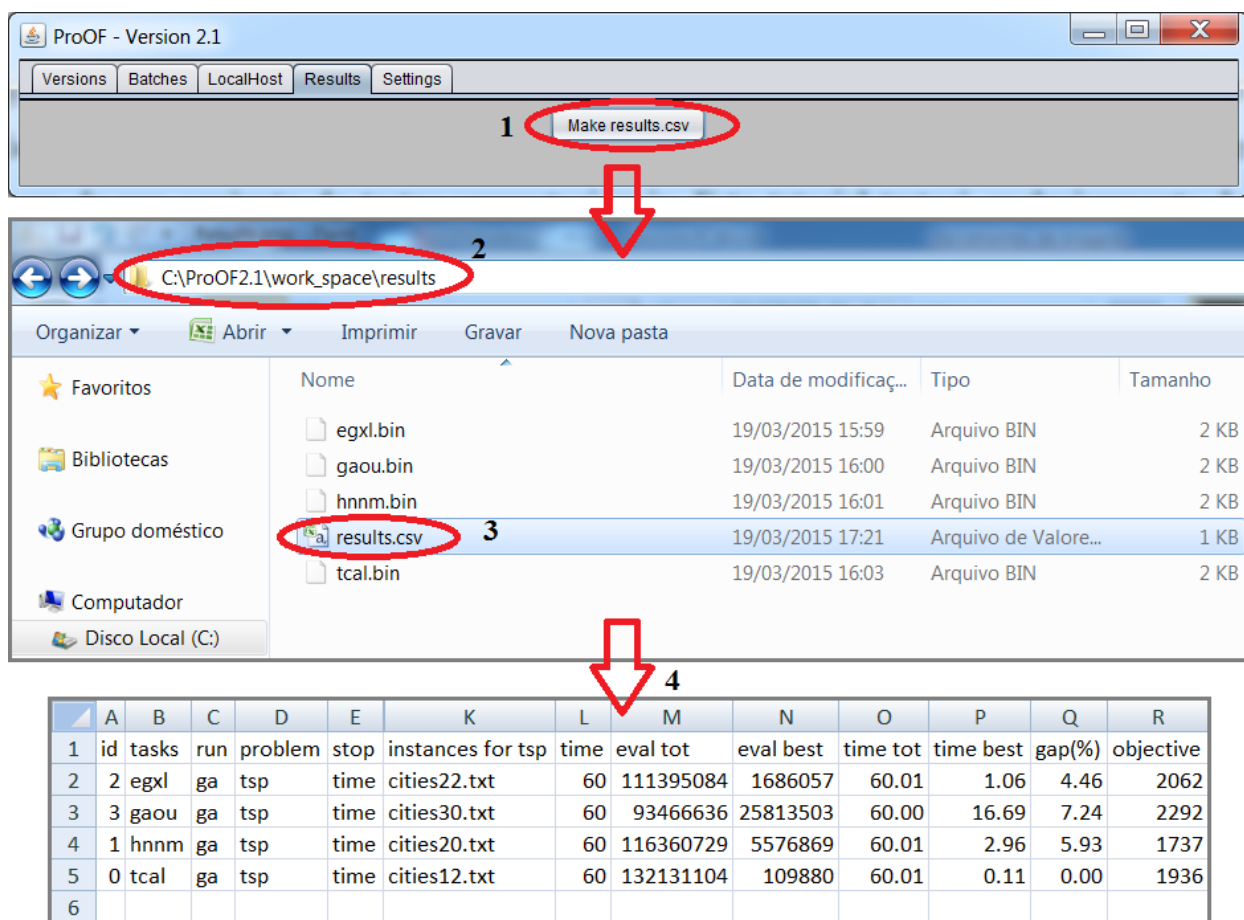


Figura 7: Visualizando todos os resultados após a execução.

2.4. Preparação para desenvolver novos códigos no ProOF

Recapitulando, o usuário vai primeiro projetar seus códigos utilizando o **ProOFJava** e utilizará o **ProOFClient** para execução dos códigos e montagem de um conjunto de testes computacionais. Este tutorial tratará exclusivamente de auxiliar o usuário na implementação de códigos para o **ProOFJava**. Assim, para iniciar o desenvolvimento de novos códigos, o usuário deve abrir o projeto **ProOFJava** utilizando a IDE NetBeans 8.0.1 (<https://netbeans.org/>).

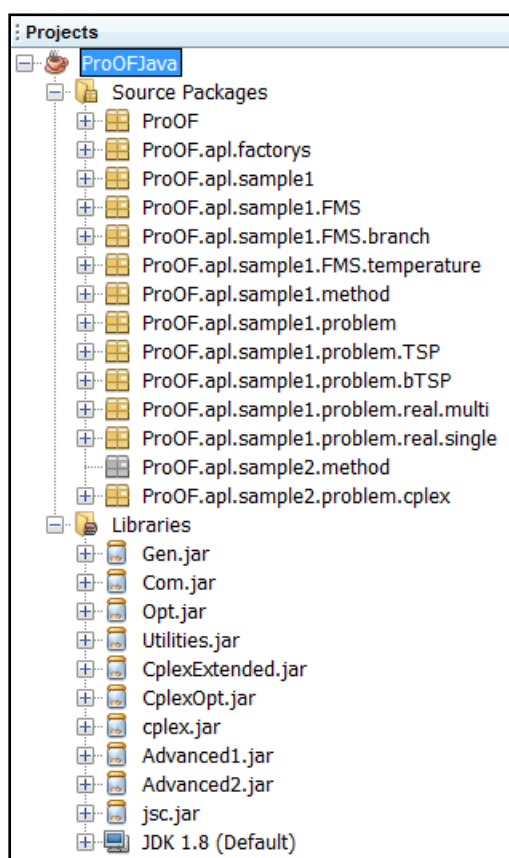


Figura 8: Projeto ProOFJava no NetBeans 8.0.1

A Figura 8 apresenta o projeto **ProOFJava** aberto no NetBeans 8.0.1. Nota-se que o projeto já vem com um conjunto de implementações como exemplo (pacote `sample1` e `sample2`), são estas implementações de exemplo que serão apresentadas ao longo deste tutorial. O usuário poderá seguir as mesmas idéias aqui apresentadas para implementar seus próprios métodos e problemas. Caso o usuário deseje já fazer suas implementações a medida que for aprendendo, sugerimos que este crie um pacote com seu próprio nome (ex: `ProOF.apl.mynome`) e siga a mesma hierarquia dos pacotes (`sample1` e `sample2`). Nota-se também que o projeto **ProOFJava** também inclui uma lista de bibliotecas. Essas bibliotecas seguem uma hierarquia de dependências conforme apresentado na Figura 9. O pacote `sample1` são para usuários que

utilizaram somente o ProOF e o pacote `sample2` para usuários que utilizaram as funcionalidades do ProOF e o solver Cplex.

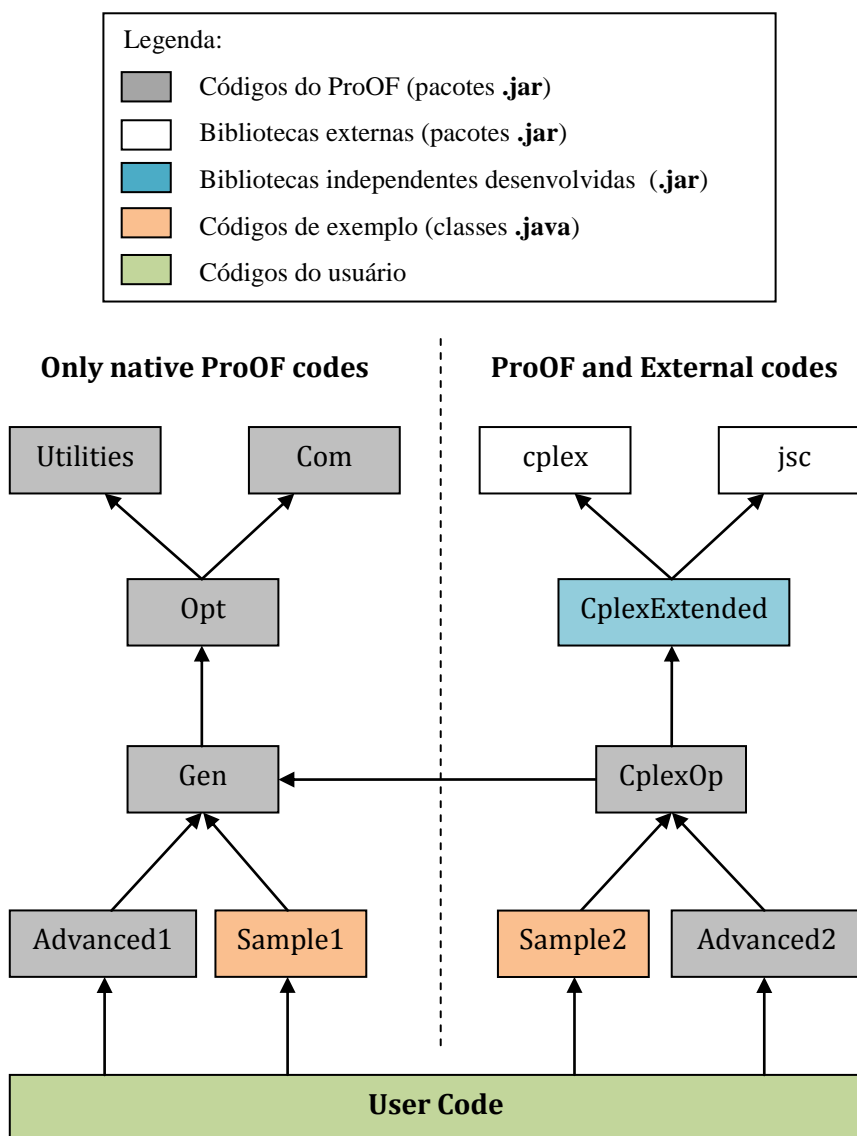


Figura 9: Hierarquia de dependência nas bibliotecas que compõem o ProOF.

Abaixo segue uma breve descrição de cada uma das bibliotecas apresentadas na Figura 9:

- **Utilities:** são um conjunto de classes que funcionam como uma simples biblioteca independente do ProOF e que fornecem várias funções úteis ao usuário.
- **Com** (communication): estabelece os protocolos de comunicação entre o código do usuário e o **ProOFClient**.
- **Opt** (optimization): estabelece as regras para incluir métodos e problemas de otimização dentro o ambiente.

- **Gen** (generic): estabelece um conjunto de componentes abstratos e gerais que podem ser reaproveitados e serão bastante úteis para o usuário como: critérios de parada, principais codificações e operadores.
- **Advanced1**: contém métodos e problemas avançados desenvolvidos no ProOF, sendo códigos considerados estáveis e disponíveis a outros usuários. Os métodos podem ser reaproveitados para resolver outros problemas do usuário e os problemas podem ser especializados para serem resolvidos por outras técnicas/métodos.
- **Sample1**: contém os métodos e problemas didáticos, servindo como exemplo no ensino de usuários introdutórios do ProOF.
- **cplex** (*solver*): biblioteca externa ao ProOF desenvolvida pela IBM Ilog contendo métodos para resolução de modelos matemáticos.
- **jsc** (*java statistical class*): biblioteca externa ao ProOF contendo funcionalidades estatísticas, utilizadas para o desenvolvimento de modelos matemáticos estocásticos dentro do ProOF.
- **CplexExtended**: trata-se de uma biblioteca independente codificada pelos desenvolvedores do ProOF e contém várias funcionalidades extras para agilizar o desenvolvimento de modelos matemáticos. Também fazendo a união do **cplex** com a **jsc** para o desenvolvimento de modelos estocásticos.
- **CplexOpt**: faz a vinculação das funcionalidades do Cplex com o ProOF permitindo aos usuários o desenvolvimento de *mathheuristics*, modelos matemáticos puros e heurísticas como *Relax&Fix* e *Fix&Optimize*.
- **Advanced2**: contém métodos e problemas avançados desenvolvidos no ProOF que trabalham com o solver Cplex. O usuário poderá reaproveitar esses métodos e problemas bem como consultar seu código como para aprender fazer o uso de estruturas avançadas do ProOF.
- **Sample2**: contém os métodos e problemas didáticos, servindo como exemplo no ensino de usuários introdutórios do ProOF e que trabalharam em conjunto com o solver Cplex.

Ao longo deste tutorial será visto com os exemplos (`sample1` e `sample2`) o uso de funcionalidades dentro de cada uma destas bibliotecas. Assim é possível começar imediatamente a implementação de métodos e problemas dentro do **ProOFJava**. Cada uma das seções seguintes será responsável pela implementação de um método ou problema no ambiente. As seções foram

organizadas em uma ordem lógica de implementação porém muitas seções podem ser lidas de forma independente. Abaixo segue a ordem para ler algumas seções:

- 3, 5, 6 e 7 são independentes
- 4 depende de 3
- 8 depende de 7
- 9 depende de 7

3. Método Mono-Objetivo: *Genetic Algorithm*

Nesta seção, serão descritos os passos necessários para inclusão no ambiente de um GA (*Genetic Algorithm*). Suponha que este GA será utilizado para resolver algum problema ainda não especificado. A Figura 10 indica onde está a classe `GeneticAlgorithm` que implementa este método de exemplo no ProOF e a Figura 11 ilustra o pseudocódigo do método que foi implementado, onde claramente não está imposta nenhuma definição específica do problema a ser resolvido.

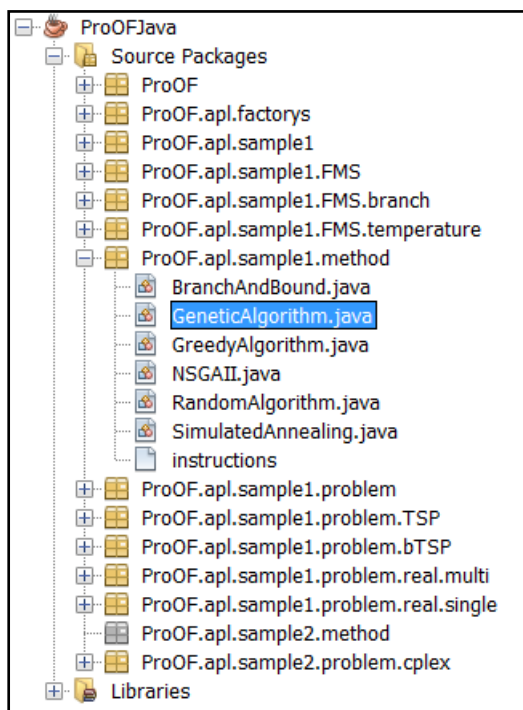


Figura 10: Classe do método Genetic Algorithm no ProOF

O primeiro passo para o usuário inserir um novo método do tipo metaheurística no ProOF é criar uma nova classe dentro do pacote `method`, a classe criada será especializada a partir da classe `MetaHeuristic`. Em seguida, dentro da nova classe haverá mais quatro passos: especializar as funções membro `name`, `services`, `parameters` e `execute`.

```

Genetic Algorithm
  Inicia a população
  Avalia a população
  repita
    Seleciona pai1 e pai2
    filho ← crossover(pai1, pai2)
    mutação(filho)
    avaliação(filho)
    inserir filho na população
  até( critério de parada ser atingido )
fim

```

Figura 11: Pseudocódigo do Genetic Algorithm implementado.

```

1  [+ ...4 lines
5  package ProOF.apl.sample1.method;
6
7  [- import ProOF.apl.factoryys.fProblem;
8     import ProOF.apl.factoryys.fStop;
9     import ProOF.com.Linker.LinkerApproaches;
10    import ProOF.com.Linker.LinkerParameters;
11    import ProOF.gen.operator.Crossover;
12    import ProOF.gen.operator.Initialization;
13    import ProOF.gen.operator.Mutation;
14    import ProOF.gen.stopping.Stop;
15    import ProOF.opt.abst.run.MetaHeuristic;
16    import ProOF.opt.abst.problem.meta.Problem;
17    import ProOF.opt.abst.problem.meta.Solution;
18    import ProOF.utilities.uTournament;
19
20  [+ /**...4 lines */
24  public class GeneticAlgorithm extends MetaHeuristic{
25      private Problem problem;
26      private Stop stop;
27      private Initialization init;
28      private Crossover cross;
29      private Mutation mut;
30
31      private int pop_size;
32      private int tour_size;
33      private double mut_rate;
34
35      @Override
36      [- public String name() {
37          return "GA";
38      }
39
40      @Override
41      [- public void services(LinkerApproaches link) throws Exception {
42          problem = link.get(fProblem.obj, problem);
43          stop = link.get(fStop.obj, stop);
44          init = link.add(Initialization.obj);
45          cross = link.add(Crossover.obj);
46          mut = link.add(Mutation.obj);
47      }
48
49      @Override
50      [+ public void parameters(LinkerParameters link) throws Exception {...5 lines }
51
52      @Override
53      [+ public void execute() throws Exception {...36 lines }
90  }

```

Figura 12: Método Genetic Algorithm no ProOF.

Assim, a Figura 12 indica como fica a estrutura do método GA inserido no ambiente, as funções membros `parameters` e `execute` foram omitidos e serão explicados adiante. Na linha 37, é atribuído um nome ao método, onde o nome definido aqui será o mesmo que ficará apresentado na interface gráfica (GUI) do módulo **Client** durante a montagem dos testes computacionais.

O terceiro passo será a vinculação de um problema, um critério de parada e dos operadores que o método irá utilizar. Para isto, o usuário declara no escopo da classe criada que haverá um problema, um critério de parada e operadores com os quais seu método irá trabalhar, como descrito nas linhas 25 – 29 na Figura 12. A especialização da função membro `services` permitirá fazer os pedidos para o ProOF que envolvem escolhas do usuário na GUI (linhas 40 – 46 na Figura 12). A função `get` é utilizada para selecionar um problema do conjunto de problemas disponíveis em `fProblem`, linha 41. Como a função `get` recebe diretamente uma referência ao objeto que contém todos os problemas já adicionados ao ambiente (`fProblem.obj`), o ProOF consegue saber entre quais problemas escolher. De forma semelhante, a função `get` é utilizada para selecionar um critério de parada (`stop`) de um conjunto de critérios de parada adicionados anteriormente ao ambiente (`fStop.obj`), linha 42. Sempre deverá ser informado para a função `get` um objeto do tipo `Factory` para que o ProOF consiga saber entre quais opções escolher. As classes do tipo `Factory` geralmente estão dentro do pacote `ProOF.apl.factorys`.

A função `add` neste exemplo é utilizada para selecionar os operadores. Esta função trabalha de forma implícita, pois não é informado para ela onde está a `Factory` que contém os operadores de inicialização, crossover e mutação. Isto ocorre porque cada operador poderá ser implementado de forma diferente para cada problema tratado. Uma representação da solução (codificação) é estabelecida para cada tipo de problema a ser solucionado. Os operadores manipulam as codificações dos problemas. Logo, só é possível escolher os operadores após a escolha de um problema. O ProOF conhece a relação entre operadores e problemas. Isso permite tratar de todos estes detalhes automaticamente e retornar ao método os operadores corretos para o problema a ser solucionado. Assim foi declarado objetos para armazenar a inicialização (`init`), o crossover (`cross`) e a mutação (`mut`) no escopo da classe, linhas 27 – 29, e a função `add` foi utilizada nas linhas 43 – 45 para que o ProOF gerencie as escolhas dos operadores indicados pelo usuário.

Ao utilizar estas funções (`get` e `add`) seguindo este exemplo, o usuário conseguirá que a interface gráfica do ProOF disponibilize uma caixa de seleção. Nesta caixa de seleção, será escolhido primeiro um dos problemas referenciados e, em seguida, uma segunda caixa de seleção permitirá ao usuário escolher qual critério de parada utilizar. Por último, caixas de seleção permitirão escolher operadores de inicialização, crossover e mutação.

Para o quarto passo será definido os parâmetros do método. Sendo assim, suponha que o usuário decidiu que seu método utilizaria seleção por torneio. Ele decidiu também que haverá três parâmetros de entrada para configurar seu método: tamanho da população (`pop_size`), tamanho da seleção (`tour_size`) e taxa de ocorrência da mutação (`mut_rate`). Valores para esses parâmetros podem ser definidos pelo usuário posteriormente na interface gráfica automaticamente gerada pelo ProOF. Logo, o quarto passo será como o usuário utilizará o ambiente para a definição destes parâmetros. O usuário deverá declarar os dois parâmetros no escopo da sua classe, linhas 31 – 33. E em seguida, sobrescrever a função membro `parameters` herdada de `MetaHeuristic`.

A Figura 13 apresenta as funções `parameters` e `execute` omitidas da figura anterior. As linhas 49 – 51, mostra como ficou a função membro `parameters` com o uso do quarto passo. Dentro da função membro `parameters` foram utilizadas funções já disponibilizadas pelo ProOF para obtenção dos parâmetros de entrada. O nome passado como argumento para a função (`link.Int`) será o nome utilizado na caixa de preenchimento disponível na GUI do ProOF. O retorno desta função será o valor escolhido na interface gráfica. Outros valores que o usuário pode solicitar ao ProOF são: Long, Float, Double e String, podendo também solicitar arquivos (File) e expressões regulares (como String).

O quinto passo será a implementação propriamente dita do método dentro do ambiente. O usuário poderá ou não utilizar as funcionalidades já incluídas no ambiente como seleção por torneio e roleta. O usuário poderá também utilizar qualquer biblioteca ou qualquer código desenvolvido por ele anteriormente para auxiliar na implementação do seu método. Assim, o usuário deve especializar a função membro `execute` e dentro dela projetar o código do seu método, linhas 54 – 89 na Figura 13. No início da execução, linhas 56 – 59, é alocada na memória uma população de soluções de tamanho `pop_size`. Na linha 62 é definido o reuso do componente `uTournament` do ProOF para tratar a seleção por torneio de tamanho `tour_size`. Em seguida todos os indivíduos são inicializados e avaliados nas linhas 65 – 68. Então, a

evolução do algoritmo começa com a seleção de dois pais (linhas 71 e 72), em seguida é criado um filho utilizando crossover (linha 75) e aplicado a mutação sobre ele (linha 79) com uma probabilidade dada por `mut_rate`. A linha 83 avalia o filho gerado.

```

47      @Override
48      public void parameters(LinkerParameters link) throws Exception {
49          pop_size = link.Int("population size", 100, 10, 10000);
50          tour_size = link.Int("tournament size", 3, 2, 16);
51          mut_rate = link.Dbl("mutation rate", 0.10, 0, 1);
52      }
53      @Override
54      public void execute() throws Exception {
55          //Declares the population and makes memory allocation
56          Solution pop[] = new Solution[pop_size];
57          for(int i=0; i<pop.length; i++){
58              pop[i] = problem.build_sol();
59          }
60
61          //Creates the managed selection by tournament
62          uTournament tour = new uTournament(pop, tour_size);
63
64          //Initiates and evaluates the population
65          for(Solution ind : pop){
66              init.initialize(ind);
67              problem.evaluate(ind);
68          }
69          do{
70              //Selection
71              int p1 = tour.select();
72              int p2 = tour.select();
73
74              //Crossover
75              Solution child = cross.crossover(pop[p1], pop[p2]);
76
77              //Mutation
78              if(Math.random() < mut_rate){
79                  mut.mutation(child);
80              }
81
82              //Evaluation
83              problem.evaluate(child);
84
85              //Insert: replaces the worst parent
86              int worse = pop[p1].GT(pop[p2]) ? p1 : p2;
87              pop[worse] = child;
88          }while(!stop.end());
89      }

```

Figura 13: Continuação do método *Genetic Algorithm* no ProOF.

A minimização da função objetivo foi adotada para este GA, onde o filho é inserido no lugar do pior pai (linhas 86 e 87) após a comparação entre p1 e p2 ser realizada. O ProOF disponibiliza um conjunto de operações para comparação do valor de duas soluções como: maior que (GT), maior ou igual a (GE), igual a (EQ), menor ou igual a (LE), menor que (LT). Por fim, a linha 88 verifica se o critério de parada foi atingido.

Após a implementação do método, o sexto e último passo a ser seguido consiste em adicionar o novo método ao conjunto de métodos do ambiente. Para isto, deve-se modificar a classe do tipo `Factory` (`fRun`), acrescentando uma instância da classe `GemectiAAlgorithm` (linha 36 da Figura 14). No projeto já está inserido como exemplo 10 métodos conforme apresentado nos índices que iniciam em 0 e vão até 9. Todas as classes dentro do pacote `ProOF.apl.factorys` apresentam esta mesma estrutura.

Dica 3.1: prevenção de erros nas classes do tipo Factory

Em todas as classes to tipo `Factory` os índices utilizados na função `build` sempre iniciam em 0 e vão até um valor N qualquer, mas os valores precisam estar em ordem crescente de 1 em 1, não podendo pular nenhum número..

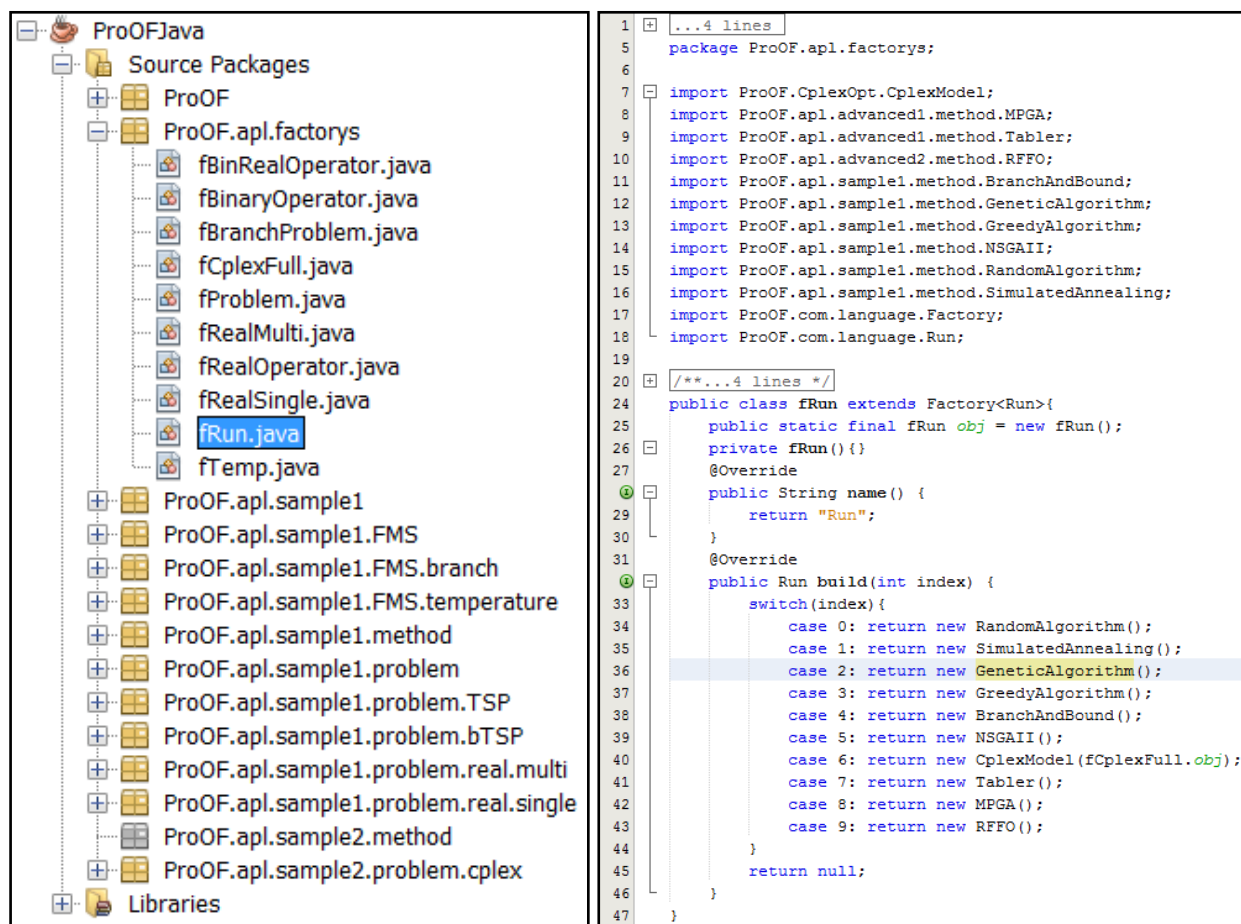


Figura 14: Modificando a classe `fRun` para adicionar o GA.

Um aspecto importante na inclusão de um método, como o GA criado, é a capacidade de ser reutilizável. Observe que em momento algum o GA foi vinculado ou conhece o problema a ser solucionado. Assim, a mesma implementação de um método será capaz de resolver vários

problemas. Essa característica ficará mais clara a seguir quando forem incluídos novos problemas nas seções 4 e 5.

Devido ao reaproveitamento do componente `Stop` (critério de parada presente nas linhas 26, 42 da Figura 12 e linha 88 da Figura 13), o ProOF deixará disponível um conjunto de critérios de parada para serem utilizados (contidos na `Factory fStop`).

Resumindo, a inclusão de um método do tipo metaheurística requer a execução dos seguintes passos:

1º Passo: criar uma nova classe especializada a partir da classe `MetaHeuristic`.

2º Passo: definir um nome para o método especializando a função membro `name`.

3º Passo: especializar a função membro `services` herdada de `MetaHeuristic` e nela realizar a vinculação do problema e dos operadores para o método.

4º Passo: definir os parâmetros do método especializando a função membro `parameters` herdada de `MetaHeuristic`.

5º Passo: implementar o código do método dentro da função membro `execute` herdada de `MetaHeuristic`.

6º Passo: modificar a classe do tipo `Factory (fRun)` para adicionar o novo método ao conjunto de métodos do ambiente.

Observe que a inclusão de um método requer a especialização de quatro funções dentro de uma única classe criada pelo usuário. Neste processo, o usuário consegue definir a entrada de parâmetros, utilizando a interface gráfica fornecida pelo ProOF, e implementar o código do método. Após isso, um ajuste na classe `fRun` permite que o método seja adicionado ao ambiente. De forma semelhante ao demonstrado com o GA, é possível incluir outros tipos de métodos metaheurísticos, heurísticos, exatos e híbridos. Neste exemplo foi necessário a implementação de apenas 90 linhas de código dentro de uma mesma classe criada pelo usuário. Mas o usuário pode decidir implementar métodos complexos e que trabalhem com mais classes e utilizem conceitos mais avançados do ProOF como será visto na seção 7 com a inclusão do método *Branch & Bound*.

4. Especializando o Caixeiro Viajante para o GA

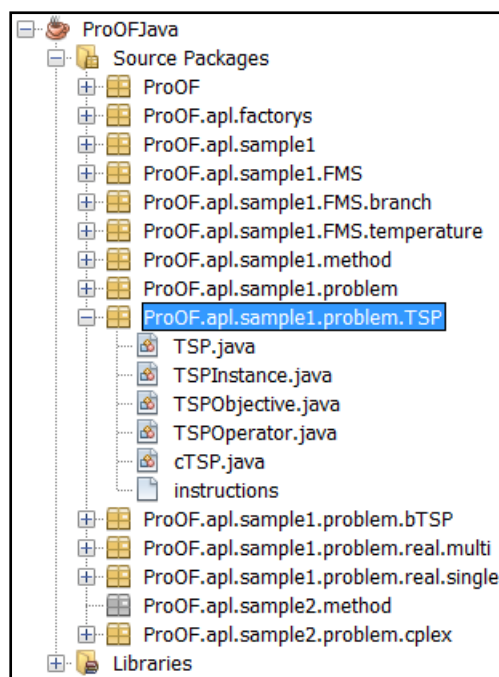


Figura 15: Pacote do problema Caixeiro Viajante no ProOF.

Agora um novo problema será incluído no ambiente: *Traveling Salesman Problem* (TSP) ou Problema do Caixeiro Viajante. Esse problema será trabalhado do ponto zero, onde haverá pouco reaproveitamento de códigos do ProOF. Afim de ilustrar os passos necessários para incluir um novo problema qualquer no ProOF, utilizando uma codificação das soluções específica para o problema e de operadores específicos para o problema. Todas as classes criadas neste exemplo para incluir o TSP no ProOF estão no pacote `ProOF.apl.sample1.problem.TSP` (Figura 15)

Assim, suponha que os dados necessários para solucionar determinada instância ou exemplar do TSP estão armazenados em um arquivo. Dada as características do problema, tal arquivo conterá uma matriz de adjacência que informa os custos das viagens entre as cidades. A Figura 16 mostra um exemplo deste arquivo considerando cinco cidades.

```

Valor_otimo
130.0
Total_de_cidades
5
Matriz_dos_custos_dos_caminhos
0      90      40      30      50
90      0      60      10      15
40      60      0      80      35
30      10      80      0      20
50      15      35      20      0

```

Figura 16: Exemplo de arquivo de entrada para o TSP.

O primeiro passo para inserir o problema no ProOF será a implementação de uma classe para ler e armazenar as informações do arquivo de entrada. Para isso, será necessária a criação de uma nova classe, chamada de `TSPInstance` (do pacote TSP), herdada da classe `Instance`, onde será implementado o código para permitir a leitura do arquivo de instância do problema. A Figura 17 descreve como ficará o código da classe `TSPInstance`.

```

1  ...4 lines
5  package ProOF.apl.sample1.problem.TSP;
6
7  import ProOF.com.Linker.LinkerParameters;
8  import ProOF.opt.abst.problem.Instance;
9  import java.io.File;
10 import java.io.FileNotFoundException;
11 import java.util.Scanner;
12
13 /**...4 lines */
17 public class TSPInstance extends Instance{
18     private File file;           //Parameter: data file problem
19
20     public int N;                 //Data: number of cities
21     public double Cij[][];       //Data: cost matrix
22     public double optimal;        //Data: known optimal value
23
24     @Override
25     public String name() {
26         return "Instance-TSP";
27     }
28     @Override
29     public void parameters(LinkerParameters link) throws Exception {
30         file = link.File("Instances for TSP", null, "txt");
31     }
32     @Override
33     public void load() throws FileNotFoundException {
34         Scanner sc = new Scanner(file);
35         sc.nextLine();
36         this.optimal = Double.parseDouble(sc.nextLine());
37         sc.nextLine();
38         this.N = Integer.parseInt(sc.nextLine());
39         sc.nextLine();
40         this.Cij = new double[N][N];
41         for(int i=0; i<N; i++){
42             for(int j=0; j<N; j++){
43                 Cij[i][j] = sc.nextDouble();
44             }
45         }
46         sc.close();
47     }
48 }

```

Figura 17: Classe para leitura de arquivo e armazenamento de instâncias do TSP.

O parâmetro `file` declarado na linha 18 é iniciado na linha 30 através de uma solicitação ao ProOF. Desta forma, o ambiente criará na interface gráfica uma opção para o usuário escolher os arquivos de instância do problema a serem solucionados. Observe que o usuário define a extensão (".txt") dentro do método `link.File` e também informa um nome que aparecerá na

GUI ("Instance for TSP"). A Figura 18 ilustra como ficará esta parte da interface gráfica. Na linha 26, o usuário informa o nome que deseja dar a este componente. A função membro `load` é chamada pelo ProOF antes do início da execução dos métodos e é utilizada aqui para ler e armazenar os dados do problema que serão utilizados durante a execução.

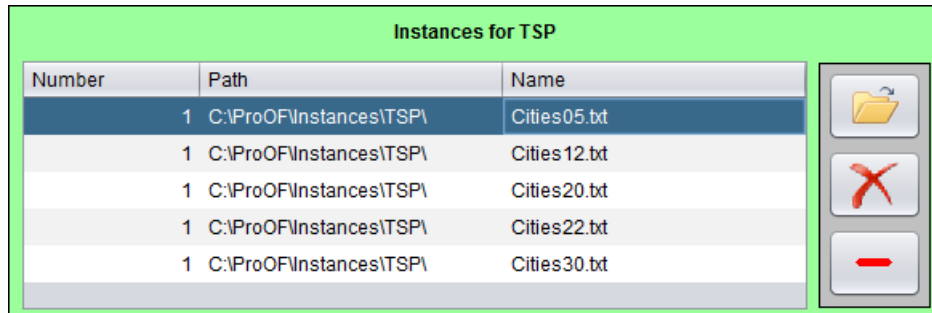


Figura 18: GUI montada pela solicitação do arquivo para o TSP.

O segundo passo é criar uma classe que representará uma solução do problema. Suponha que o usuário definiu que uma solução do problema será representada como um vetor de valores inteiros contendo os índices das cidades percorridas pelo caixeiro. Logo, a representação da solução (codificação) será uma permutação de N elementos do conjunto $[0 \dots N-1]$. A Figura 19 exemplifica uma solução dado os custos definidos na Figura 16.

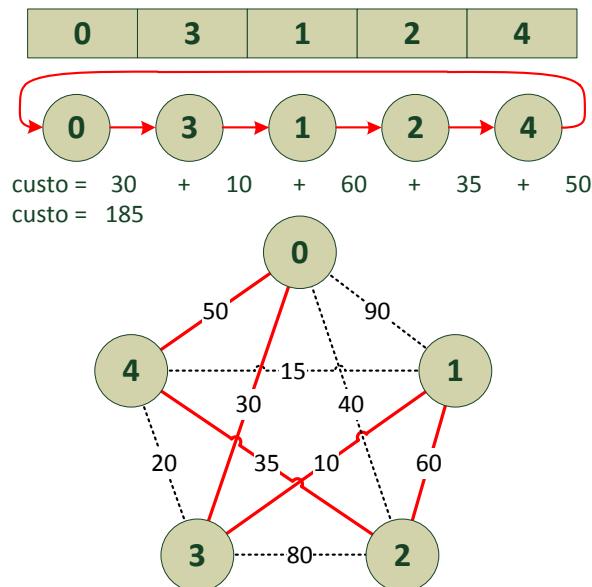


Figura 19: Exemplo de representação da solução (codificação) e sua rota (decodificação).

A Figura 20 contém a classe `cTSP` criada para implementar a representação da solução proposta para o TSP. Essa classe estende a classe abstrata `Codification`. Na linha 14 é declarado o vetor (`path`) que armazenará as cidades percorridas. No construtor, linha 17, o vetor é alocado com tamanho N (dados definido anteriormente em `TSPInstance`).

Dica 4.1: mensagens de erro na IDE

O ProOF faz forte uso de templates para fazer estruturas genéricas e robustas como as classes: Codification, Objective, Operator e Problem. Assim na linha 13 da Figura 20 é preciso informar para a nova codificação criada a classe do problema a ser resolvido (TSP) e a própria classe criada (cTSP). A classe TSP será a ultima classe criada neste exemplo pois ela é responsável por ligar todas as demais. Assim muitas IDEs acusarão um erro em todos os locais que for declarado a classe TSP pois ela ainda não existe. Isto não é um problema pois ainda não terminamos de definir tudo a respeito do problema, também não será possível compilar o código enquanto não terminar a implementação completa do problema.

```

1  ...4 lines
5  package ProOF.apl.sample1.problem.TSP;
6
7  import ProOF.opt.abst.problem.meta.codification.Codification;
8
9  /**...4 lines */
13 public class cTSP extends Codification<TSP, cTSP> {
14     protected int path[];
15
16     public cTSP(TSP prob) {
17         this.path = new int[prob.inst.N];
18     }
19     @Override
20     public void copy(TSP prob, cTSP source) throws Exception {
21         System.arraycopy(source.path, 0, this.path, 0, this.path.length);
22     }
23     @Override
24     public cTSP build(TSP prob) throws Exception {
25         return new cTSP(prob);
26     }
27 }

```

Figura 20: Codificação do problema TSP.

Basicamente, a implementação da classe de codificação para o problema deve conter: a estrutura de dados para representar a solução e as funções membro `copy` e `build`. Estas funções membro são utilizadas pelo ProOF, respectivamente para fazer cópias e novas alocações de memória da codificação implementada.

O terceiro passo é definir a função objetivo para avaliar as soluções do problema. A Figura 21 ilustra a classe que define a função objetivo do TSP. Como se trata de um problema mono-objetivo, a classe `TSPObjective` especializa `SingleObjective` e deve implementar as funções membro `evaluate` e `build`. A primeira função é responsável por calcular o custo da codificação e a segunda é utilizada para alocar novos objetivos. As linhas de 18 – 26 calculam o custo do percurso. A função membro `set` (linha 25) armazena o valor do custo encontrado. A

função membro `build` é utilizada pelo ProOF para alocar mais objetos desta mesma classe (linha 29).

Dica 4.2: erros comuns na função `copy` da classe `codification`

Um erro que ocorre com frequência ProOF é na implementação da função membro `copy`. Alerta-se que dentro deste método deve-se fazer a cópia de toda a informação declarada no escopo da classe. Assim é comum que o usuário ao longo do desenvolvimento de sua codificação acabe acrescentando mais estruturas para armazenar a codificação do seu problema e esqueça de fazer a cópia desta nova estrutura. Isto é um erro grave pois o erro é de lógica e não será detectado pelo ambiente e fará com que os métodos funcionem mal pois com frequência fazem cópias das soluções e assim suas boas soluções ficarão deterioradas pois perderão parte da informação de suas codificações. Outro erro comum dentro desta função é copiar apenas os ponteiros de um objeto e não copiar o conteúdo do objeto. É preciso lembrar que em Java todos os tipos não básicos são referenciados por ponteiros implícitos, assim fazer com que um objeto receba outro não fará cópia e sim com que duas codificações referenciem um mesmo objeto. Isto também é um erro grave, por exemplo: em algoritmos que trabalham com buscas locais uma solução vizinha é definida ao fazer uma cópia da solução corrente e em seguida fazer uma pequena perturbação na sua codificação. Assim uma perturbação no objeto que não foi copiado corretamente perturbará a solução corrente do algoritmo e fará com que o algoritmo tenha um desempenho ruim por causa desse erro que será difícil de detectar.

```

1  ...4 lines
5  package ProOF.apl.sample1.problem.TSP;
6
7  import ProOF.opt.abst.problem.meta.objective.SingleObjective;
8
9  /**...4 lines */
13 public class TSPObjective extends SingleObjective<TSP, cTSP, TSPObjective> {
14     public TSPObjective() throws Exception {
15         super();
16     }
17     @Override
18     public void evaluate(TSP prob, cTSP codif) throws Exception {
19         double fitness = 0;
20         int i = codif.path[codif.path.length-1];    //last city
21         for(int j : codif.path){
22             fitness += prob.inst.Cij[i][j];
23             i = j;
24         }
25         set(fitness);    //set de fitness to the ProOF
26     }
27     @Override
28     public TSPObjective build(TSP prob) throws Exception {
29         return new TSPObjective();
30     }
31 }

```

Figura 21: Função objetivo do problema TSP.

```

1  [+ ...4 lines
5  package ProOF.apl.sample1.problem.TSP;
6
7  [- import ProOF.com.language.Factory;
8  import ProOF.gen.operator.oCrossover;
9  import ProOF.gen.operator.oInitialization;
10 import ProOF.gen.operator.oLocalMove;
11 import ProOF.gen.operator.oMutation;
12 import ProOF.opt.abst.problem.meta.codification.Operator;
13 import java.util.Random;
14
15  [+ /**...4 lines */
19  public class TSPOperator extends Factory<Operator>{
20      public static final TSPOperator obj = new TSPOperator();
21
22      @Override
23      public String name() {
24          return "TSP Operators";
25      }
26      @Override
27      public Operator build(int index) { //build the operators
28          switch(index){
29              case 0: return new RandomTour(); //initialization
30              case 1: return new MutExchange(); //mutation
31              case 2: return new TwoPoints(); //crossover
32              case 3: return new MovExchange(); //local movement
33          }
34          return null;
35      }
36
37      [+ private class RandomTour extends oInitialization<TSP, cTSP>{...15 lines }
52      [+ private class MutExchange extends oMutation<TSP, cTSP>{...10 lines }
62      [+ private class MovExchange extends oLocalMove<TSP, cTSP>{...10 lines }
72      [+ private class TwoPoints extends oCrossover<TSP, cTSP>{...27 lines }
99
100 [- private static void random_swap(Random rmd, cTSP ind){
101     int a = rmd.nextInt(ind.path.length);
102     int b = rmd.nextInt(ind.path.length);
103     int aux = ind.path[a];
104     ind.path[a] = ind.path[b];
105     ind.path[b] = aux;
106 }
107 }

```

Figura 22: Classe Factory para acesso aos operadores do TSP.

O quarto passo é implementar os operadores que manipulam a codificação do problema. No exemplo, serão implementados os operadores de inicialização, crossover e mutação. Existem formas diferentes de se fazer isto, mas deixaremos aqui uma sugestão de acreditamos ser mais apropriada. Assim será criada a classe `TSPOperator` filha de `Factory` (Figura 22). É definido um único objeto estático para um objeto desta classe e com acesso publico, linha 20. Um nome é dado a este novo componente, linha 24, e a função `build` é definida para instanciar cada um dos operadores desta `Factory`, linhas 25 – 35. As implementações de cada um dos operadores é definida em classes privadas começando nas linhas 37, 52, 62 e 72 da Figura 22. As linhas 100 – 106 definem uma função estática (`random_swap`) que será utilizada pelos operadores de

inicialização, movimento local e mutação. Esta função sorteia duas posições (a e b) aleatoriamente do circuito as trocam de posição. Adiante será apresentada cada uma das classes criadas para cada um dos operadores.

Dica 4.3: definindo mais que um operador do mesmo tipo

Neste exemplo vemos na função `build` que é instanciado apenas um operador de cada tipo (`oInitialization`, `oMutation`, `oLocalMove` e `oCrossover`). Mas é possível e recomendado que o usuário defina mais que somente uma implementação de cada um. Assim posteriormente o usuário poderá escolher no momento da execução dos testes computacionais qual de cada tipo será utilizado ou até se todo o conjunto será utilizado. Como exemplo o usuário pode olhar a classe `fRealOperator` do pacote `ProOF.apl.factorys`, nela a função `build` constrói 10 operadores de crossover, 3 mutações, 3 movimentos locais e 1 inicialização para a codificação real já disponível no `ProOF`.

O primeiro operador implementado é o operador de inicialização (`RandomTour`). A Figura 23 ilustra este operador que faz a geração de uma permutação aleatória. A classe `RandomTour` especializa a classe `oInitialization`, onde deve ser informado um nome para o operador e sobrescrever a função membro `initialize`. A inicialização implementada garante que a codificação seja uma solução factível, pois gera uma permutação sem repetição de cidades.

```

37 private class RandomTour extends oInitialization<TSP, cTSP>{
38     @Override
39     public String name() {
40         return "Random Tour";
41     }
42     @Override
43     public void initialize(TSP prob, cTSP ind) throws Exception {
44         for(int i=0; i<ind.path.length; i++){
45             ind.path[i] = i;
46         }
47         for(int i=0; i<ind.path.length; i++){
48             random_swap(prob.rnd, ind);
49         }
50     }
51 }

```

Figura 23: Operador de inicialização para o TSP.

A Figura 24 demonstra o operador de mutação. A classe `MutExchange` especializa a classe `oMutation` e deve informar um nome para o operador e sobrescrever a função membro `mutation`. Para este operador foi adotado uma pequena perturbação na solução, trocando as posições de duas cidades aleatoriamente escolhidas (com a função `random_swap` definida na linha 100 da Figura 22). Seguindo a mesma ideia da mutação foi implementado também o

operador de movimento local, que não será escrito agora pois não é um requisito para resolver o TSP com o GA.

```

52  private class MutExchange extends oMutation<TSP, cTSP>{
53      @Override
54      public String name() {
55          return "Mut-Exchange";
56      }
57      @Override
58      public void mutation(TSP prob, cTSP ind) throws Exception {
59          random_swap(prob.rnd, ind);
60      }
61  }

```

Figura 24: Operador de mutação para o TSP.

A Figura 25 ilustra um exemplo do operador de crossover que será implementado para o TSP. A classe criada especializa a classe `oCrossover` e deve informar um nome para o operador e sobrescrever a função membro `crossover`. No exemplo considerado, primeiramente são selecionados dois pontos de corte. O filho (`child`) receberá do primeiro pai (`ind1`) todo o percurso entre os pontos (cidades 3 e 1). Em seguida, a partir do final do segundo ponto de corte, o filho receberá todas as cidades do segundo pai (`ind2`) que não foram selecionadas para o filho ainda (cidades 0, 4 e 2). Esse crossover garante que a solução seja válida para o problema já que também gera uma permutação sem repetições.

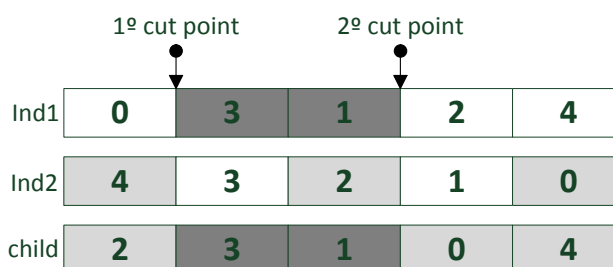


Figura 25: Exemplo do crossover de dois pontos para o TSP.

Assim, a Figura 26 mostra como ficou o código do operador de crossover. Inicialmente, o novo espaço de memória para uma nova codificação (`child`) é alocado (linha 79) utilizando a função `build` definida anteriormente para a codificação `cTSP`. Em seguida, dois pontos de cortes (linha 80) são selecionados e o filho recebe do primeiro pai (`ind1`) toda a representação que vai do ponto de corte `p[0]` até `p[1]` (linhas 82 até 85). Depois, partindo do ponto de corte `p[1]` até o ponto de corte `p[0]` (fechando o ciclo), o filho recebe as cidades que não foram escolhidas ainda do segundo pai (linhas 86 até 95). Por fim, a nova codificação preenchida é retornada (linha 96).

```

72 private class TwoPoints extends oCrossover<TSP, cTSP>{
73     @Override
74     public String name() {
75         return "TwoPoints";
76     }
77     @Override
78     public cTSP crossover(TSP prob, cTSP ind1, cTSP ind2) throws Exception {
79         cTSP child = ind1.build(prob);
80         int p[] = prob.rnd.cuts_points(prob.inst.N, 2);
81         boolean selected[] = new boolean[prob.inst.N];
82         for(int i=p[0]; i<p[1]; i++){
83             child.path[i] = ind1.path[i];
84             selected[child.path[i]] = true;
85         }
86         int i=p[1];
87         int j=p[1];
88         while(i!=p[0]){
89             while(selected[ind2.path[j]]){
90                 j = (j+1) % prob.inst.N;
91             }
92             child.path[i] = ind2.path[j];
93             selected[child.path[i]] = true;
94             i = (i+1) % prob.inst.N;
95         }
96         return child;
97     }
98 }

```

Figura 26: Operador de crossover para o TSP.

Dica 4.4: copia errada dos genes dos pais para o filho durante o crossover

Também é um erro comum e grave fazer a copia apenas da referencia para um dado gene durante o crossover. Assim como no método `copy` comentado na dica 4.2, se a codificação dos genes de um problema é feita por objetos então será preciso copiar explicitamente os genes recebidos de `ind1` ou de `ind2` durante o crossover. Caso contrário uma mutação futura no gene do filho também modificará o gene do pai que o transmitiu. Sendo assim faça atribuições diretas tais como nas linhas 83 e 92 da Figura 26, apenas para variáveis de tipos básicos em java (`int`, `float`, `double`, `boolean`, ...) para os demais tipos (classe em Java) sugerimos criar um construtor de cópia ou uma função membro para fazer a copia explicita do objeto.

O quinto passo é a criação da classe `TSP` responsável por ligar as diferentes partes do problema (Figura 27). Assim foi definido: um nome para o problema (linha 22), a construção de novas codificações para o TSP (linha 26) e construção de novos objetivos (linha 30). Também foi habilitado o acesso aos operadores do TSP (linha 36) e a classe que contém o arquivo de entrada do TSP (linha 35). Descrevendo melhor, a linha 30 aceita retornos de classes para os tipos `SingleObjective` e `MultiObjective`, que podem ser utilizados para problemas mono e multi-objetivo, respectivamente. Na linha 40, deve-se retornar o objeto do componente `BestSol` para todos os problemas mono-objetivos e para problemas multi-objetivos retorna-se `null`. A

função `start` é de implementação opcional, aqui neste exemplo ela foi implementada para informar ao ProOF que a instância atual do TSP possui solução ótima conhecida e que deseja-se que o ProOF imprima uma coluna extra de informação durante a execução do método informando o *gap* da solução corrente em relação a este ótimo.

```

1  [+ ...4 lines
5  package ProOF.apl.sample1.problem.TSP;
6
7  [- import ProOF.com.Linker.LinkerApproaches;
8  import ProOF.gen.best.BestSol;
9  import ProOF.opt.abst.problem.meta.Objective;
10 import ProOF.opt.abst.problem.meta.Problem;
11 import ProOF.opt.abst.problem.meta.codification.Codification;
12
13 [+ /**...4 lines */
17 public class TSP extends Problem<BestSol>{
18     public final TSPInstance inst = new TSPInstance();
19
20     @Override
21     ① [- public String name() {
22         return "TSP";
23     }
24     @Override
25     ② [- public Codification build_codif() throws Exception {
26         return new cTSP(this);
27     }
28     @Override
29     ③ [- public Objective build_obj() throws Exception {
30         return new TSPObj();
31     }
32     @Override
33     ④ [- public void services(LinkerApproaches link) throws Exception {
34         super.services(link);
35         link.add(inst);
36         link.add(TSPOperator.obj);
37     }
38     @Override
39     ⑤ [- public BestSol best() {
40         return BestSol.object();
41     }
42     @Override
43     ⑥ [- public void start() throws Exception {
44         add_gap("gap", inst.optimal);
45     }
46 }

```

Figura 27: Classe para ligar as diferentes partes do problema TSP.

O sexto e último passo é adicionar o problema TSP criado ao conjunto de problemas do ambiente. Para isso, o novo problema será adicionado à classe `fProblem`. A Figura 28 ilustra como ficou a classe `fProblem` e a inclusão do TSP (linha 29). A partir deste momento já é possível resolver o caixeiro viajante utilizando o algoritmo genético.

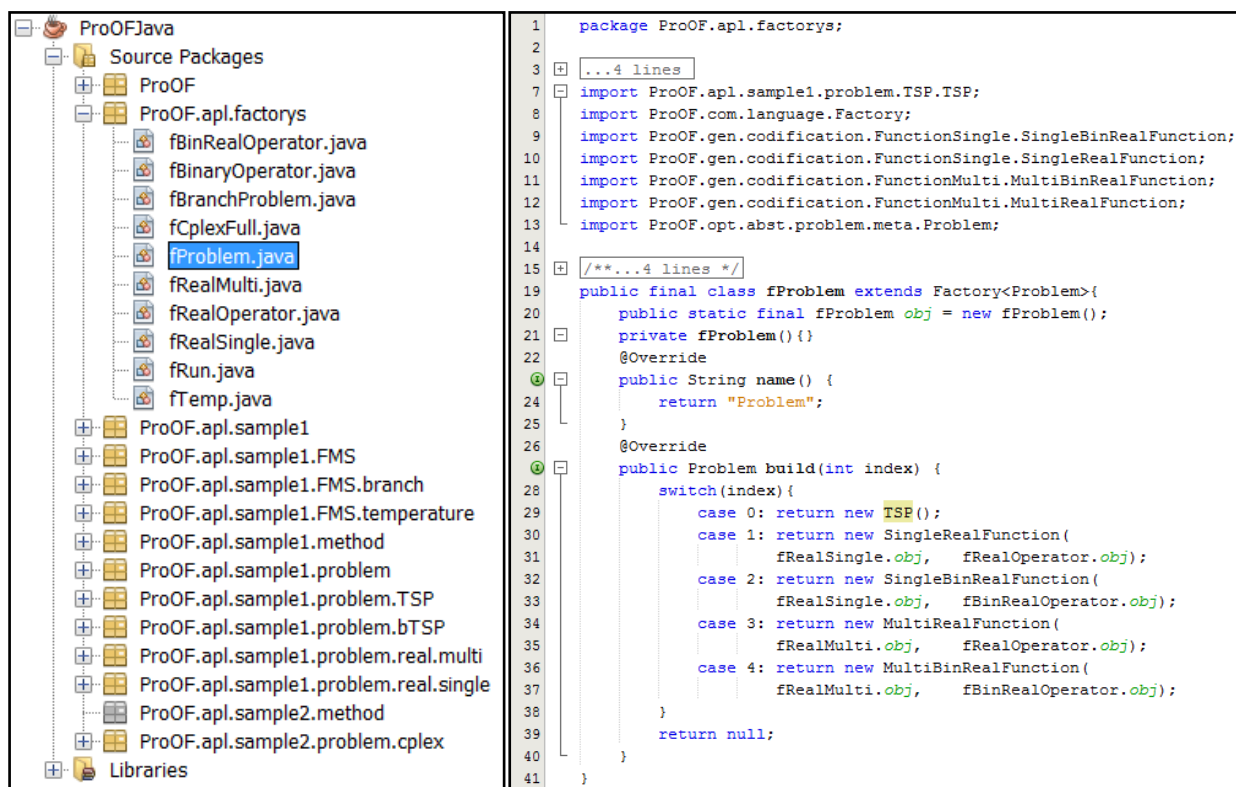


Figura 28: Incluindo TSP na classe Problem.

Assim, foi demonstrado acima como incluir o problema TSP no ambiente e aqui será feita uma breve revisão dos passos seguidos.

1º Passo: criar uma nova classe para a leitura de dados do problema: TSPInstance.

2º Passo: definir uma classe para representar uma solução do problema: cTSP.

3º Passo: uma classe para calcular/armazenar o custo da solução: TSPObjective.

4º Passo: definir uma classe Factory (TSPOperator) e implementar operadores para o problema.

5º Passo: especializar a classe TSP como filha de Problem e nela realizar a vinculação com as demais classes que definem o problema.

6º Passo: modificar a classe do tipo Factory (fProblem) para adicionar o novo problema ao conjunto de problemas do ambiente.

Vemos que diferente do método foi necessário a criação de diversas classes para especializar o problema. Assim na seção 5 a seguir, incluiremos a função multimodal ACK que precisará da especialização de apenas uma classe pois reaproveitará a codificação real já definida no ProOF.

5. Função Multimodal Mono-Objetivo

A fim de facilitar a inclusão de problemas com reuso de codificação real e seus operadores uma nova classe abstrata, `RealSingle`, foi criada para que o usuário possa especializar determinada função de domínio real. Neste caso, deve ser informado apenas um nome (`name`), número de variáveis no domínio (`size`) e o cálculo da função objetivo (`F`). Assim implementaremos a função ACK com $x \in \mathbb{R}^n$ descrita na equação (1) e para acrescentar esta função no ambiente, deve-se primeiro implementar a classe ACK (Figura 29).

$$f(x) = 20 + e - 20 \exp \left(-2 \sqrt{\frac{\sum_{i=1}^n \left(\frac{x_i}{100} \right)^2}{n}} \right) - \exp \left(\frac{\sum_{i=1}^n \cos(2\pi x_i)}{n} \right) \quad (1)$$

$$-30 \leq x_i \leq +30$$

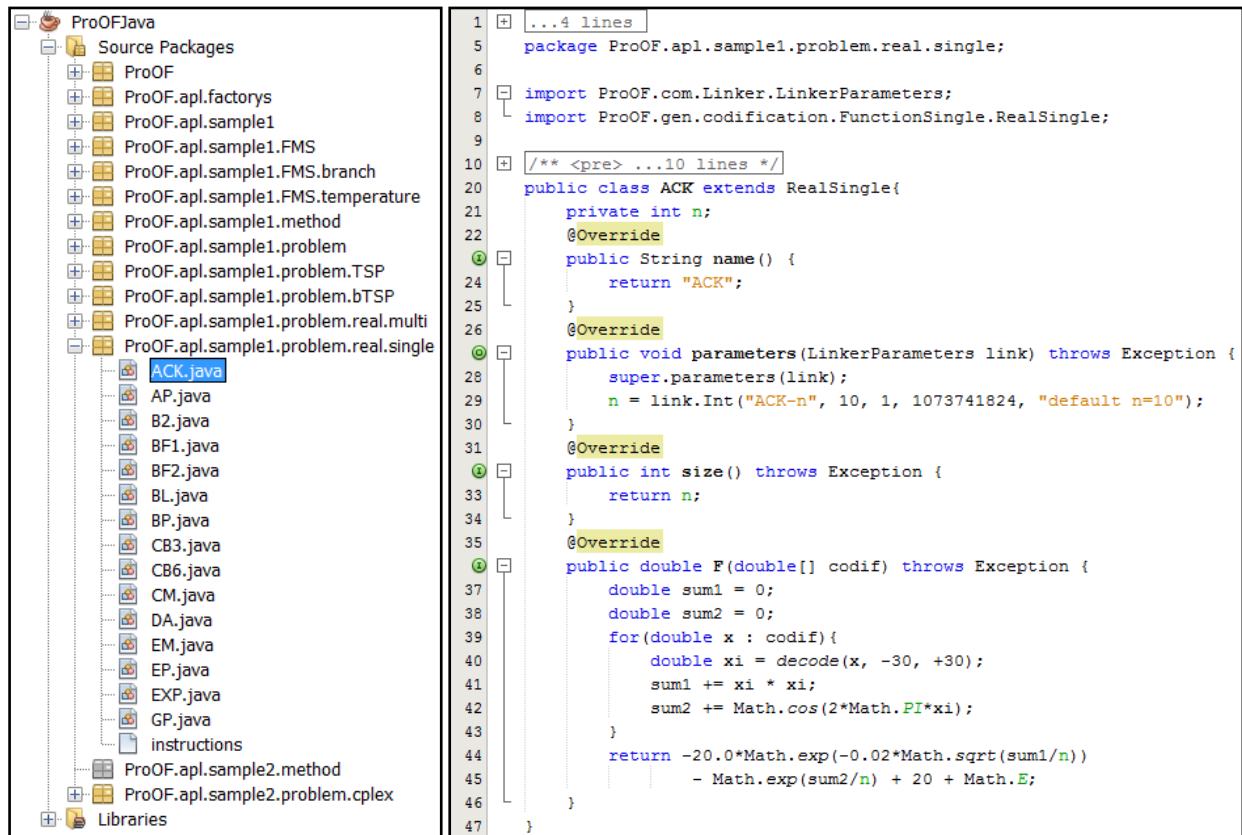


Figura 29: Implementação da classe ACK.

A classe criada especializa a classe `RealSingle` e deve informar um nome para a função (`name`), o número de variáveis no domínio (`size`) e sobrescrever a função membro `F`. A linha 24 define um nome para a função e o número de variáveis ou dimensões no espaço é definido na linha 33. Note que o valor `n` será definido pelo usuário no momento dos testes, pois na linha 29 é

informado ao ProOF que este parâmetro deve ser estabelecido. Em seguida, é descrita a função F onde uma codificação `codif` com n valores reais no intervalo de $[0, 1]$ é passada como argumento e na linha 40 é utilizada a função `decode` para recuperar o valor x_i entre $[-30, +30]$.

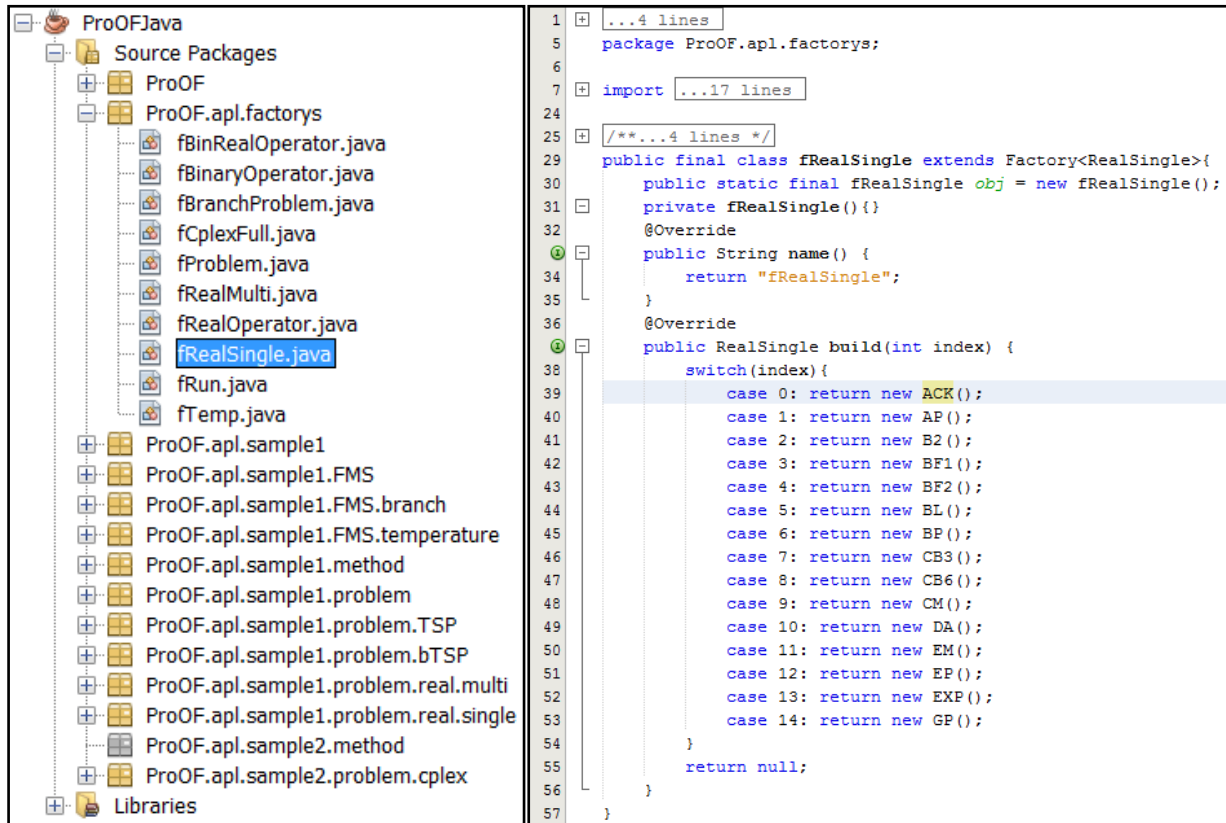


Figura 30: Incluindo ACK na classe `fRealSingle`.

O segundo e último passo para incluir a função é modificar a classe `fRealSingle` para que o ambiente reconheça mais este problema, Figura 30.

6. Modelando o TSP com o uso do Cplex no ProOF

Essa seção será feita no futuro.

7. Especializando o TSP ser resolvido pelo RFFO

Essa seção será feita no futuro.

8. Introdução aos aspectos avançados do ProOF

Essa seção será feita no futuro e deverá conter: como criar novas estruturas FMS, desenvolvimento de modelos matemáticos combinados com metaheurísticas, criação de novos operadores, métodos e problemas multiobjetivos.

9. Implementando o método exato Branch&Bound

Essa seção será feita no futuro.

10. Especializando o TSP para o Branch&Bound

Essa seção será feita no futuro.

11. Implementando uma heurística gulosa genérica

Essa seção será feita no futuro.

12. Implementando o método NSGA-II para problemas multiobjetivo

Essa seção será feita no futuro.

13. Definindo uma função multiobjetivo

Essa seção será feita no futuro.

14. Conclusões

Essa seção será feita no futuro.