ProOF

Tutorial Rápido de introdução à ferramenta

Autores: Márcio da Silva Arantes, André Missaglia e Marcelo Hossomi.

Visão geral

Esta ferramenta adota uma estrutura que torna fácil a implementação de problemas de otimização, fazendo amplo uso de conceitos como reutilização de código e gerenciamento de interfaces.

Para diversas aplicações, é interessante que haja um método simples para obtenção de parâmetros, leitura de dados (instancias) ou execução distribuída de métodos em várias máquinas. Além disso, deve ser facilitada também a compatibilidade entre códigos já implementados. Por exemplo, a ideia de um algoritmo genético é a mesma para problemas distintos como o *Travelling Salesman Problem-TSP* (Problema do caixeiro viajante) ou otimização de funções multimodais.

O mesmo vale para diferentes métodos aplicados a um problema. A codificação de um problema de funções multimodais, bem como o código para operadores de crossover e mutação, permanecem idênticos se queremos comparar o desempenho de dois algoritmos diferentes.

O ProOF é dividido em *Abstract*, *Client*, *Serve* e *Slave*. Os módulos *Server* e *Slave* são usados para execução distribuída em várias máquinas. *Client* é a interface para seleção de parâmetros e execução da aplicação. *Abstract* é a biblioteca do ProOF que deve ser incluída no próprio projeto do usuário.

Este tutorial aborda apenas a implementação de um método, um problema, e as aplicações *Client* e *Abstract*, deixando de lado tanto a estrutura interna quanto a utilização de funções que seriam úteis no desenvolvimento de uma aplicação avançada.

Incluindo um novo método

Iniciaremos com a inclusão de um algoritmo genético, que será utilizado futuramente para um problema ainda não especificado.

```
Algoritmo Genético
     Inicia a população;
     Avalia a população;
     repita
           Seleciona pai1 e pai2;
           filho ← crossover(pai1, pai2);
           mutação(filho);
           avaliação(filho);
           inserir filho na população;
     até( criterio de parada ser atingido )
fim
```

Figura 1: Pseudocódigo de um algoritmo genético simples.

O Framework é composto basicamente por *Nós*. Neste caso, definimos o algoritmo genético, o problema, e os operadores como nós do sistema, e todo o relacionamento entre estes nós é o que torna esta ferramenta poderosa.

Suponha o seu nó como o algoritmo em si. Para o AG funcionar, é necessário um problema e um critério de parada. O critério de parada pode ser por número de iterações ou tempo decorrido.

Aqui surge uma restrição relevante do sistema: o AG implementado só executa sobre um problema que tenha operadores de inicialização, crossover e mutação implementados para este problema. Veremos mais a frente que para isso é necessário executar um `get` para problemas e critérios de parada, e um `need` para os operadores citados.

Dentre as possibilidades de nós, existe um tipo específico chamado `Run`. Todas as classes que herdam de `Run` possuem o método `execute()`. Desta forma é possível que este nó seja executado.

Para fins de organização, existe a classe `MetaHeuristic`, filha de `Run`. É sobre esta que daremos o primeiro passo.

[GeneticAlgorithm.h]

```
1   #ifndef GENETICALGORITHM_H
2   #define GENETICALGORITHM_H
3
4   #include "MetaHeuristic.h"
5
6   class GeneticAlgorithm : public MetaHeuristic{
7   public:
8       GeneticAlgorithm();
9       virtual ~GeneticAlgorithm();
10
11      virtual const char* name()const;
12      virtual const char* description()const;
13  private:
14
15  };
16
17  #endif
```

[GeneticAlgorithm.cpp]

```
18  #include "GeneticAlgorithm.h"
19
20  const char* GeneticAlgorithm::name()const{
21      return "GA";
22  }
23  const char* GeneticAlgorithm::description()const{
24      return NULL;
25  }
```

Figura 2: Inserindo um método

O usuário deverá criar os arquivos *.h e *.cpp como exemplificado na Figura 2 para o AG: GeneticAlgorithm.h e GeneticAlgorithm.cpp. Neste ponto o usuário já tem dois arquivos para inserção da sua implementação do AG.

Para que o framework reconheça a classe criada, é necessário adicioná-la ao factory `fRun`:

```
----------------------------------[fRun.cpp]-----------------------------------
1    #include "fRun.h"
2    #include "GeneticAlgorithm.h"
3
4    const fRun fRun::obj;
5
6    const char* fRun::name()const{
7        return "Run";
8    }
9    Node* fRun::NewService(int index)const{
10       switch(index){
11           case 0 : return new GeneticAlgorithm();
12       }
13       return NULL;
14   }
```

Figura 3: Modificando a classe fRun

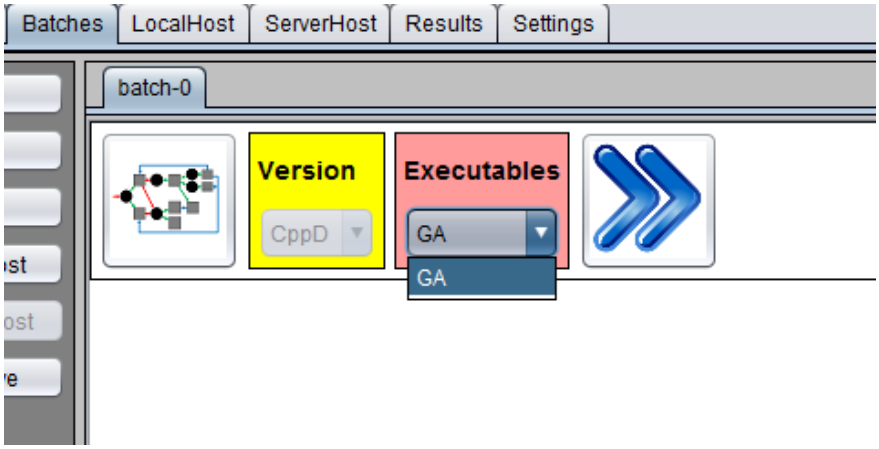

Figura 4: Nó disponível para execução na interface gráfica

Todo nó permite o gerenciamento de parâmetros e serviços. Em linhas gerais um parâmetro define um atributo do nó, enquanto que um serviço define os relacionamentos deste com outros nós.

Por exemplo, suponha que o usuário decidiu que haverá dois parâmetros de entrada para configurar seu método: tamanho da população (`pop_size`) e máximo de avaliações (`max_eval`). Esses parâmetros podem ser definidos na interface gráfica automaticamente gerada pelo *ProOF*.

Logo, o segundo passo será a definição dos parâmetros de entrado do método em desenvolvimento pelo usuário no *ProOF*. Inicialmente, o usuário deverá declarar os dois parâmetros no escopo da sua classe. Isso ocorre nas linhas 13 e 14 no código GeneticAlgorithm.h da Figura 5. Em seguida, o usuário sobrescreve a função membro `parameters` herdada de `MetaHeuristic`, permitindo assim que a entrada de dados seja feita via interface gráfica gerada pelo *ProOF*. Isso foi feito nas linhas 23 a 26 do arquivo GeneticAlgorithm.cpp na Figura 5.

Dentro da função membro `parameters` foram utilizadas funções já disponibilizadas pelo *ProOF* para obtenção de parâmetros de entrada. Um objeto do tipo `ParameterLinker` é utilizado para isso. Os argumentos passados para as funções (`Int` e `Dbl`) são respectivamente: um nome para o parâmetro, seu valor padrão e seus valores mínimo e máximo permitidos. O retorno destas funções será o valor escolhido na interface gráfica. Outros valores que o usuário pode solicitar com este objeto são : Long, Float, String, podendo também solicitar arquivos e expressões regulares.

----------[GeneticAlgorithm.h]----------

```
1   #ifndef GENETICALGORITHM_H
2   #define GENETICALGORITHM_H
3
4   #include "MetaHeuristic.h"
5
6   class GeneticAlgorithm : public MetaHeuristic{
7   public:
8       .
9       .
10      .
11      virtual void parameters(LinkerParameters *link) throw(exception);
12  private:
13      int pop_size;
14      int max_eval;
15  };
16
17  #endif
```

----------[GeneticAlgorithm.cpp]----------

```
18  #include "GeneticAlgorithm.h"
19  #include "LinkerParameters.h"
20      .
21      .
22      .
23  void GeneticAlgorithm::parameters(LinkerParameters *link) throw(exception){
24      pop_size = link->Int("numero de individuos", 10);
25      max_eval = link->Int("numero de avaliacoes", 100, 10, 1000000);
26  }
```

Figura 5 : Obtendo os parâmetros.

O terceiro passo será a vinculação do problema e dos operadores que o método irá utilizar. Para isto, o usuário declara no escopo da classe criada o tipo de problema e operadores com o qual seu método vai trabalhar, sobrescrevendo a função membro `services` da classe `MetaHeuristic`. Esse passo é ilustrado pelas linhas 15 a 21 na Figura 6, onde foram utilizadas as classes abstratas Problem, oInitializer, oCrossover e oMutation para definir os ponteiros para os operadores a serem desenvolvidos pelo usuário (init, cross e mut). A Figura 6 apresenta como ficou o terceiro passo no código.

```
----------------------------[GeneticAlgorithm.h]-----------------------------
1   #ifndef GENETICALGORITHM_H
2   #define GENETICALGORITHM_H
3
4   #include "MetaHeuristic.h"
5   #include "Problem.h"
6   #include "oInitializer.h"
7   #include "oCrossover.h"
8   #include "oMutation.h"
9
10  class GeneticAlgorithm : public MetaHeuristic{
11  public:
12      .
13      .
14      .
15      virtual void services(LinkerNodes* link) throw(exception);
16  private:
17      Problem* problem;
18      oInitializer* init;
19      oCrossover* cross;
20      oMutation* mut;
21  };
22
23  #endif
----------------------------[GeneticAlgorithm.cpp]----------------------------
24  #include "GeneticAlgorithm.h"
25  #include "LinkerParameters.h"
26  #include "LinkerNodes.h"
27  #include "fProblem.h"
28      .
29      .
30      .
31  void GeneticAlgorithm::services(LinkerNodes *link) throw(exception){
32      problem = link->get(&fProblem::obj, problem);
33      init    = link->need(init);
34      cross   = link->need(cross);
35      mut     = link->need(mut);
36  }
```

Figura 6: Vinculação do problema e dos operadores

Dentro da função membro `services`, um objeto do tipo `ServicerLinker` é utilizado para selecionar o problema e os operadores. Aqui a função `get` é utilizada para selecionar um problema do conjunto de problemas que lhe é fornecido como primeiro parâmetro. A função `get` recebe diretamente uma referência ao objeto que contém todos os problemas já adicionados ao ambiente (`Problems.obj`). De forma semelhante, função `get` é utilizada para selecionar um critério de parada (`stop`) de um conjunto de critérios de parada adicionados anteriormente ao ambiente (`Stops.obj`).

A função `need` neste exemplo é utilizada para selecionar os operadores definidos pelo usuário, onde seu argumento é o próprio ponteiro para o objeto. Desta forma, o usuário deve implementar as funções get e need seguindo como exemplo o que foi feito nas linhas 32 a 35 da Figura 6. Estas funções trabalham de forma implícita, pois é informado para elas apenas o tipo do operador (`oInitializer`, `oCrossover`, `oMutation`).

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 soluções dos problemas. Logo, só é possível escolher os operadores após a escolha de um problema. O *ProOF* conhece a relação entre operadores e problemas, permitindo tratar de todos estes detalhes automaticamente e retornando ao método os operadores corretos para o problema a ser solucionado.

De maneira simplificada: É necessária a escolha de um problema, o qual estará cadastrado em uma *Factory* especificada. Neste caso usamos a função `get`, pois sabemos exatamente onde estão os nós desejados. Os problemas estão limitados a fornecer de alguma forma os operadores necessários. Para estes, usamos a função `need`, uma vez que não sabemos onde eles se encontram.

O quarto passo será a implementação propriamente dita do algoritmo genético dentro do ambiente. Esse passo é ilustrado pelas Figuras 7 e 8. O usuário poderá ou não utilizar as funcionalidades disponíveis no ambiente como gerenciadores de crossover, mutação, seleção por torneio e roleta. O *ProOF* possui estas abstrações prontas, mas é possível herdar ou criar novas classes totalmente independentes, caso as abstrações já existentes não sirvam para o propósito desejado. O usuário poderá também utilizar qualquer biblioteca da linguagem escolhida ou qualquer código desenvolvido por ele anteriormente.

[GeneticAlgorithm.h]

```
1   #ifndef GENETICALGORITHM_H
2   #define GENETICALGORITHM_H
3
4   #include "MetaHeuristic.h"
5   #include "Problem.h"
6   #include "oInitializer.h"
7   #include "oCrossover.h"
8   #include "oMutation.h"
9   #include "Solution.h"
10
11  class GeneticAlgorithm : public MetaHeuristic{
12  public:
13      .
14      .
15      .
16      virtual void execute() throw(exception);
17  private:
18      .
19      .
20      .
21      int count_eval;
22      int tour(Solution** pop);
23      void evaluate(Solution* sol);
24  };
25
26  #endif
```

Figura 7: Implementação do algoritmo genético no ambiente: Cabeçalho.

```
----------------------------[GeneticAlgorithm.cpp]----------------------------
1   #include "GeneticAlgorithm.h"
2   #include "LinkerParameters.h"
3   #include "LinkerNodes.h"
4   #include "fProblem.h"
5   #include "Solution.h"
6   #include <cstdlib>
7         .
8         .
9         .
10  void GeneticAlgorithm::execute() throw(exception){
11      count_eval = 0;
12      //Alocação de memória
13      Solution** pop = new Solution*[pop_size];
14      for(int i=0; i<pop_size; i++){
15          pop[i] = problem->NewSolution();
16      }
17
18      //Inicia a população
19      for(int i=0; i<pop_size; i++){
20          init->initialize(problem, pop[i]);
21      }
22
23      //Avalia a população
24      for(int i=0; i<pop_size; i++){
25          evaluate(pop[i]);
26      }
27
28      do{
29          //seleciona dois pais
30          int p1 = tour(pop);
31          int p2 = tour(pop);
32
33          //filho = crossover (pais)
34          Solution* child = cross->crossover(problem, pop[p1], pop[p2]);
35          //mutaçao(filho)
36          mut->mutation(problem, child);
37          //avaliar(filho)
38          evaluate(child);
39
40          //insere na populacao
41          int worse = *(pop[p1])>*(pop[p2]) ? p1 : p2;
42          delete pop[worse];
43          pop[worse] = child;
44      }while(count_eval < max_eval);
45  }
46
47  int GeneticAlgorithm::tour(Solution** pop){
48      int i = rand()%pop_size;
49      int j = rand()%pop_size;
50
51      if(*(pop[i])<*(pop[j])){
52          return i;
53      }else{
54          return j;
55      }
56  }
57
58  void GeneticAlgorithm::evaluate(Solution* sol){
59      problem->evaluate(sol);
60      count_eval++;
61  }
```

Figura 8 : Implementação do algoritmo genético no ambiente: Código.

Incluindo um novo problema

Um aspecto importante na inclusão de um método, como o algoritmo genético criado, é a capacidade de ser reutilizável. Observe que em momento algum o algoritmo genético 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. Para isso, utilizaremos como exemplo o problema de otimização da função seno dada por:

$$f(x) = x \sin(10\pi x) + 1$$

$$-1 \leq x \leq 2$$

Será utilizada a codificação binária por simplicidade. Uma possível solução S seria:

$$S = 1010101001_B$$

A conversão da solução para um valor de x dentro do seu domínio é:

$$x(S) = \min + (\max - \min)\frac{S}{2^{10} - 1}$$

$$x(S) = 3\frac{S}{2^{10} - 1} - 1$$

Para tanto começamos definindo nossa codificação, o crossover, mutação, e função objetivo, já herdando funcionalidades do ProOF:

Para representar a codificação, vamos criar os arquivos Binary.h e Binary,cpp. A classe Binary deverá ser filha da classe Codification como ilustrado na linha 6. Além disso, deve implementar as funções Binary (construtor), Copy e New como ilustrado nas linhas 8 a 13. Além disso, foi declarado um vetor de inteiros para armazenar os valores binários (linha 15). Esse vetor (ponteiro) é o único código criado pelo usuário.

```
----------------------------[ Binary.h]-----------------------------
1   #ifndef BINARY_H
2   #define BINARY_H
3
4   #include "Codification.h"
5
6   class Binary : public Codification{
7   public:
8       Binary();
9       virtual ~Binary();
10
11      virtual void Copy(Problem* prob, Codification* source)
           throw(exception);
12
13      virtual Codification* New(Problem* prob) throw(exception);
14
15      int* cromo;
16  private:
17
18  };
19
20  #endif
---------------------------[ Binary.cpp]----------------------------
21  #include "Binary.h"
22
23  Binary::Binary() {
24      cromo = new int[10];
25  }
26  Binary::~Binary() {
27      delete cromo;
28  }
29  void Binary::Copy(Problem *prob, Codification *source)
    throw(exception){
30      Binary* codif = dynamic_cast<Binary*>(source);
31      for(int i=0; i<10; i++){
32          this->cromo[i] = codif->cromo[i];
33      }
34  }
35  Codification* Binary::New(Problem *prob) throw(exception){
36      return new Binary();
37  }
```

Figura 9 : Codificação binária implementada no *ProOF*

```
----------------------------[ SINObjective.h]-----------------------------
1   #ifndef SINOBJECTIVE_H
2   #define SINOBJECTIVE_H
3
4   #include "SingleObjective.h"
5
6   class SINObjective : public SingleObjective{
7   public:
8       SINObjective();
9       virtual ~SINObjective();
10
11      virtual void Evaluate(Problem* mem, Codification* codif)
            throw(exception);
12
13      virtual Objective* New(Problem* mem) throw(exception);
14
15  private:
16      double decode(int *cromo, int size, double min, double max);
17  };
18
19  #endif
----------------------------[ SINObjective.cpp]----------------------------
20  #include "SINObjective.h"
21  #include "Binary.h"
22  #include <math.h>
23  #define PI 3.14159265
24
25  SINObjective::SINObjective() {
26  }
27
28  SINObjective::~SINObjective() {
29  }
30
31  void SINObjective::Evaluate(Problem* prob, Codification* A)
    throw(exception){
32      Binary* codif = dynamic_cast<Binary*>(A);
33
34      double x = decode(codif->cromo, 10, -1, +2);
35
36      double fitness = x*sin(10*PI*x) + 1;
37
38      set(fitness);
39  }
40
41  Objective* SINObjective::New(Problem* prob) throw(exception){
42      return new SINObjective();
43  }
44
45  double SINObjective::decode(int *cromo, int tam, double min, double max){
46      double b10 = 0;
47      double pot = 1;
48      for(int i=0; i<tam; i++){
49          b10 += cromo[i]*pot;
50          pot = pot*2;
51      }
52      return min + (max-min)*b10/(pot-1);
53  }
```

Figura 10: Função objetivo implementada no *ProOF*

O próximo passo é implementar a função objetivo como ilustrado na Figura 10. Como o problema em questão possui apenas um objetivo (apenas um valor a ser avaliado), a classe de função objetivo implementada pelo usuário deve estender a classe `SingleObjective` e

implementar os métodos `Evaluate` e `New`: Observe que a função objetivo é a própria função seno definida anteriormente.

O próximo passo é implementar os operadores que manipulam a codificação do problema. Nesse exemplo, implementaremos um operador de inicialização, um de crossover e um de mutação. Para isso, criamos o arquivo BinaryOperator.h, onde estão definidas a classe BinaryOperator, filha de Factory, como ilustrado na Figura 11.

```
----------------------------[ BinaryOperator.h]-----------------------------
1   #ifndef BINARYOPERATOR_H
2   #define BINARYOPERATOR_H
3
4   #include "Factory.h"
5
6   class BinaryOperator : public Factory{
7   public:
8       static const BinaryOperator obj;
9       BinaryOperator();
10      virtual ~BinaryOperator();
11
12      virtual const char* name()const;
13      virtual Node* NewService(int index)const;
14
15  private:
16      class INIT;
17      class CROSS;
18      class MUT;
19  };
20
21  #endif
```

Figura 11 : Cabeçalho da classe BinaryOperator

A seguir apresentamos a implementação de um operador de inicialização simples (classe `INIT`) que consiste em gerar uma codificação binária aleatoriamente, como ilustrado nas linhas 18 a 28 da Figura 12. O operador de crossover deste exemplo será o uniforme (classe `CROSS`), que consiste em escolher, bit a bit, um dos pais para herdar com 50% de chance para cada. A implementação está descrita nas linhas 29 a 46 da Figura 12. Finalmente, o operador de mutação do exemplo inverterá um único bit da codificação (classe `MUT`) como apresentado nas linhas 47 a 56 da Figura 12.

Para que os operadores sejam reconhecidos pelo framework, é preciso implementar a classe responsável por criá-los. Essa classe será um singleton (`BinaryOperator`), isto é, existirá apenas um objeto dela (linha 8 da figura 11 com a linha 9 da figura 12), e deverá extender a classe Factory.

----------[**BinaryOperator.cpp**]----------

```
1   #include "BinaryOperator.h"
2   #include "Problem.h"
3   #include "Binary.h"
4   #include "oCrossover.h"
5   #include "oMutation.h"
6   #include "oInitializer.h"
7   #include <cstdlib>
8
9   const BinaryOperator BinaryOperator::obj;
10
11  BinaryOperator::BinaryOperator() {
12  }
13  BinaryOperator::~BinaryOperator() {
14  }
15  const char* BinaryOperator::name()const{
16      return "Bin-Operator";
17  }
18  class BinaryOperator::INIT: public oInitializer{
19      const char* name()const{
20          return "Bin-init";
21      }
22      void initialize(Problem* prob, Codification* ind) throw(exception){
23          Binary* codif = dynamic_cast<Binary*>(ind);
24          for(int i=0; i<10; i++){
25              codif->cromo[i] = rand()%2;
26          }
27      };
28  };
29  class BinaryOperator::CROSS: public oCrossover{
30      const char* name()const{
31          return "Bin-cross";
32      }
33      Codification* crossover(Problem* prob, Codification* ind1,
           Codification* ind2) throw(exception){
34          Binary* codif1 = dynamic_cast<Binary*>(ind1);
35          Binary* codif2 = dynamic_cast<Binary*>(ind2);
36          Binary* child  = dynamic_cast<Binary*>(codif1->New(prob));
37          for(int i=0; i<10; i++){
38              if(rand()%2){
39                  child->cromo[i] = codif1->cromo[i];
40              }else{
41                  child->cromo[i] = codif2->cromo[i];
42              }
43          }
44          return child;
45      };
46  };
47  class BinaryOperator::MUT: public oMutation{
48      const char* name()const{
49          return "Bin-mut";
50      }
51      void mutation(Problem* prob, Codification* ind) throw(exception){
52          Binary* codif = dynamic_cast<Binary*>(ind);
53          int i = rand()%10;
54          codif->cromo[i] = 1 - codif->cromo[i];
55      };
56  };
57  Node* BinaryOperator::NewService(int index)const{
58      switch(index){
59          case 0: return new INIT();
60          case 1: return new CROSS();
61          case 2: return new MUT();
62      }
63      return NULL;
64  }
```

Figura 12 : Implementação da classe BinaryOperator com três operadores

Agora, deve-se definir a classe `SINProblem`. Ela possui apenas métodos para criar todas as outras classes, devendo extender a classe `Problem`. Desta forma, o usuário deverá implementar exatamente como descrito na Figura 13.

----------[SINProblem.h]----------

```
1   #ifndef SINPROBLEM_H
2   #define SINPROBLEM_H
3
4   #include "Problem.h"
5
6   class SINProblem : public Problem{
7   public:
8       SINProblem();
9       virtual ~SINProblem();
10
11      virtual const char* name() const;
12      virtual const char* description() const;
13      virtual Codification *NewCodification() throw(exception);
14      virtual Objective *NewObjective() throw(exception);
15      virtual void services(LinkerNodes *com) throw(exception);
16  private:
17
18  };
19
20  #endif
```

----------[SINProblem.cpp]----------

```
21  #include "SINProblem.h"
22  #include "Binary.h"
23  #include "SINObjective.h"
24  #include "LinkerNodes.h"
25  #include "BinaryOperator.h"
26
27  SINProblem::SINProblem() {
28  }
29
30  SINProblem::~SINProblem() {
31  }
32
33  const char* SINProblem::name() const{
34      return "SIN";
35  }
36  const char* SINProblem::description() const{
37      return NULL;
38  }
39  Codification* SINProblem::NewCodification() throw(exception){
40      return new SINCodification();
41  }
42  Objective* SINProblem::NewObjective() throw(exception){
42      return new SINObjective();
43  }
44  void SINProblem::services(LinkerNodes *link) throw(exception){
45      link->addf(&SINOperator::obj);
46  }
```

Figura 13 : Implementação da classe BinaryOperator com três operadores

Por fim, deve-se incluir o novo problema no ambiente, modificando a classe fProblem como mostrado na Figura 14.

```
--------------------------------[ fProblem.cpp]--------------------------------
1    #include "fProblem.h"
2    #include "SINProblem.h"
3
4    const fProblem fProblem::obj;
5
6    const char* fProblem::name()const{
7        return "Problem";
8    }
9    Node* fProblem::NewService(int index)const{
10       switch(index){
11           case 0 : return new SINProblem();
12       }
13       return NULL;
14   }
```

Figura 14 : Modificando a classe fProblem

Utilizando a interface:

O primeiro passo para testar a aplicação é catalogá-la no sistema. Para isso, basta selecionar a linguagem utilizada e sua localização no sistema. O Framework se encarregará de compilar e adicionar os componentes no sistema.

Figura 15: Catalogando o código novo

Com o código no sistema, é necessário criar um batch para a execução do código. Primeiro clicamos em New, selecionamos a versão do código e começamos a selecionar os nós necessários. Conforme os nós são selecionados, o grafo com os componentes e serviços é atualizado. Para visualizar o grafo, basta clicar no botão ao lado da caixa de seleção "Version".

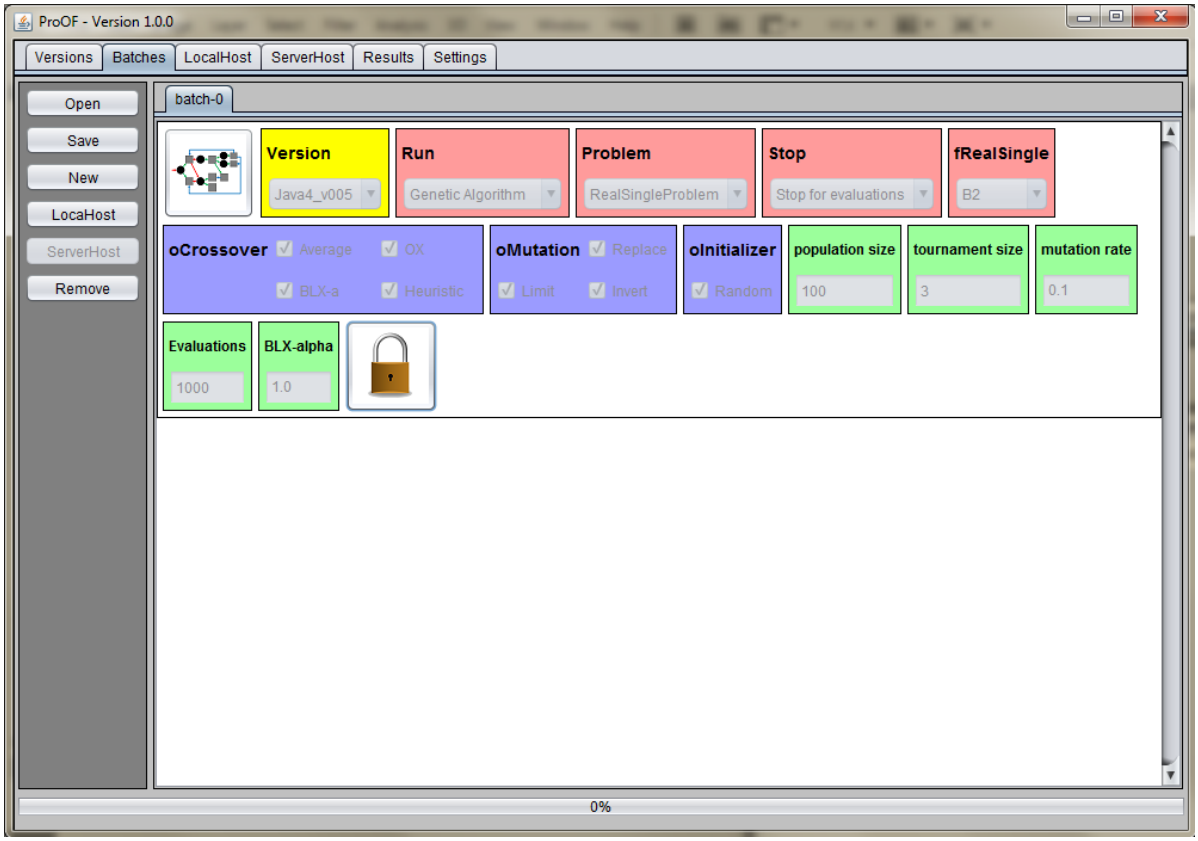

Figura 16: Montando um batch para a execução do código

A seleção de componentes ocorre de maneira bastante intuitiva, assim como a entrada de parâmetros que ocorre em seguida. Após isso bloqueamos o batch e clicamos em "Localhost" para execução na própria máquina.

Na aba "LocalHost" aparecem todos os *jobs* criados até o momento. Um *job* é uma execução do programa. Até o momento deve ter aparecido um único *job*, mas é possível que um *batch* gere mais de um *job*.

Observe que o *job* criado possui o estado "waiting", o que significa que ele será executado. Primeiramente precisamos salvar os *jobs*, e depois executá-los, ambos clicando no 4º e 5º botão da barra lateral esquerda, respectivamente.

É possível também adicionar mais de uma thread para execução dos *jobs*. Cada *job* será executado em uma thread.

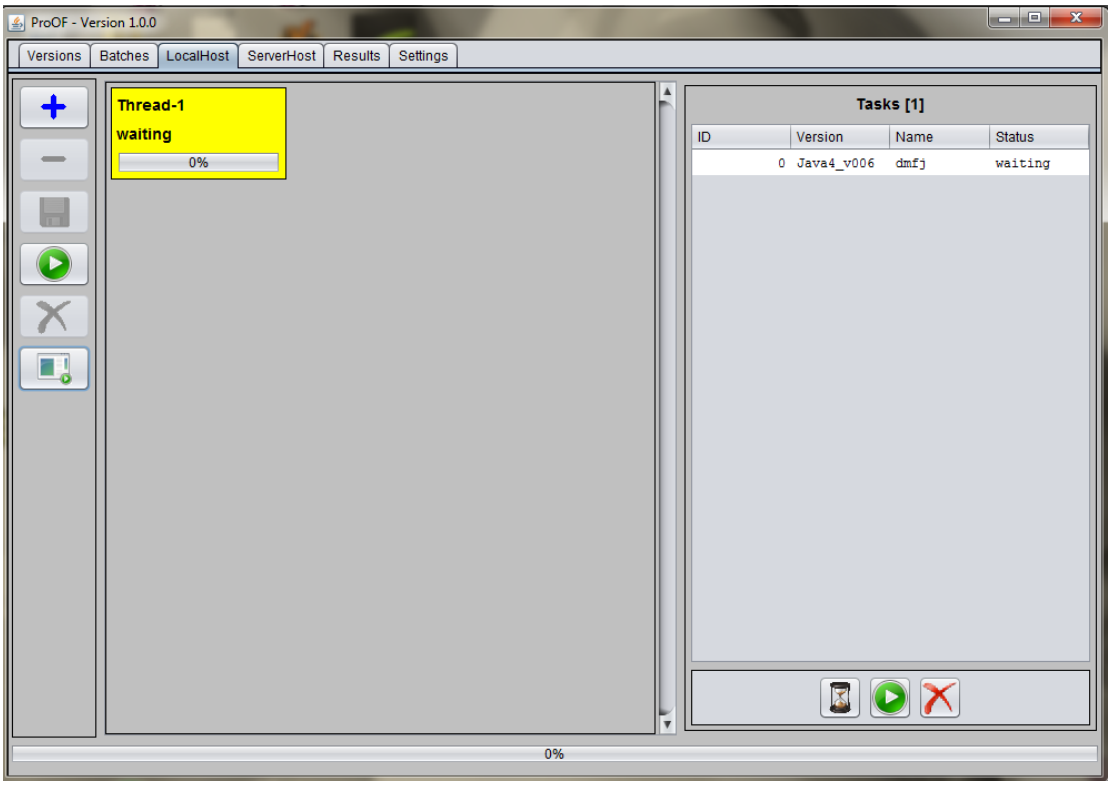

Figura 17: Preparando para a execução do código

Clicando no último botão da barra lateral, é possível conferir as saídas do seu programa. Estas saídas serão salvas em *workspace/run/Thread-<N>/<job>.out*. Caso seu código produza algum resultado, ele será salvo nos arquivos *.wrt* , o qual podemos usar para gerar uma tabela com os resultados de todas as execuções na aba "Results". É possível também criar mais de um canal de saída, assim como uma barra de progresso, se estes forem solicitadas em algum local do código.

Apêndice I – Padrões e Práticas

Padrões de nomes

As classes utilizadas seguem um padrão específico para alguns casos, com uma letra minúscula na frente do nome usado. Ex.: `oCrossover`. Estas letras podem ser:

- `a` – Abstração.
- `f` – Factory de nós.
- `o` – Operador.
- `i` – Implementação de uma abstração (`a`).
- `p` – Um nó auxiliar, o qual será utilizado (`need`) por um determinado nó, mas será gerenciado (fornecido, `add`) por outro nó qualquer.
- `n` – Um nó auxiliar, o qual será gerenciado (fornecido, `add`) por um determinado nó, mas poderá será utilizado (`need`) por outro qualquer.

Algumas práticas

- Nunca execute um numero maior de threads do que a quantidade de núcleos de sua máquina.
- Caso haja um erro de compilação ao catalogar o seu código, remova a pasta *workspace/* e reinicie o cliente, para evitar erros futuros. Se os erros persistirem, envie os arquivos proof.log e compiler.log para o seguinte e-mail: marcio.da.silva.arantes@gmail.com.
- Ao terminar um batch, principalmente se ele exigir uma quantidade grande de parâmetros, salve o batch. Isto é útil para testes ou mesmo para salvar a configuração que fez o algoritmo ter a melhor configuração.
- Não será necessário executar o cliente para cada modificação feita. Neste caso, modifique a linha onde é especificado o job a ser executado, no arquivo principal (main, figura 18 abaixo) do *abstract*.

```
53  int main(int argc, char** argv) {
54      //Client version
55      client(argc, argv);
56
57      //##### para testes do modelo descomente a linha abaixo ######
58      //model();
59
60      //##### para testes da execução descomente a linha abaixo ######
61      //run("D:\\ProOF\\work_space\\waiting\\job", "D:\\ProOF\\work_space\\input\\");
62
63      return 0;
64  }
```

Figura 18: main.cpp