# Documentação: Programação genética para o problema de regressão simbólica

**Larissa Fernandes Leijôto**

[1]Departamento de Ciência da Computação – Universidade Federal De Minas Gerais (UFMG)
Caixa Postal 6627 – 31270-901 – Minas Gerais – MG – Brasil

larissaleijoto@ufmg.br
Computação Natural - 2/2015

***Resumo.*** *O problema de regressão simbólica, consiste na manipulação de expressões matemáticas com o objetivo de encontrar a melhor função que descreve um conjunto de pontos. Esta documentação tem a finalidade de apresentar, discorrer e analisar a resolução do problema de regressão simbólica por meio de uma algoritmo de programação genética. Mostraremos como o problema foi modelado, a implementação, as instruções para a execução e análise das saídas geradas pelo algoritmo.*

## 1. Introdução

A programação genética é uma ramificação dentro da computação evolutiva, na qual é baseada na teoria da evolução de Darwin. Ela é uma meta-heurística que realiza um processo de busca e otimização inspirada em recombinação genética. Essa busca parte de uma população inicial, no qual cada indivíduo é associado a uma solução em potencial dentro do conjunto de soluções. Cada indivíduo possui um valor de *fitness* que determina quanto ele está adaptado ao ambiente.

## 2. Modelagem e solução proposta

Nesta seção será discutida a modelagem dos indivíduos, bem como a sua avaliação. Apresentaremos também como os operadores de cruzamentos mutação foram implementados e as dificuldades enfrentadas durante sua implementação.

### 2.1. Considerações iniciais

O algoritmo implementado neste trabalho segue o fluxo básico, de um algoritmo evolucionário, sendo assim se inicia em um processo de seleção, no qual se baseia na *fitness*). Os indivíduos escolhidos para permanecerem na população são então recombinados através dos operadores genéticos, cruzamentos e mutação. A partir daí, o processo se repete, esperando-se obter um melhor valor de *fitness* a cada população gerada. O pseudo código 1 ilustra o fluxo principal do algoritmo implementado.

**Entrada**: Conjunto de pontos
**Saída**: Melhor indivíduo
1 Inicializar população aleatória;
2 **repita**
3 | avaliar os indivíduos da população;
4 | executar seleção;
5 | executar cruzamento;
6 | executar mutação;
7 **até** *(Alcançar critério de convergência)*;

**Algoritmo 1:** Esboço do algoritmo implementado

## 2.2. Indivíduo

Para facilitar a criação de expressões matemáticas, utilizamos uma árvore. Um indivíduo é composto pela sua *fitness* e a sua árvore. Essa árvore é composta de nodos, que podem ser **terminais**, **coeficientes** e **operadores**. Um exemplo de uma árvore e sua expressão em pré-ordem podem sr visto na Figura 1.

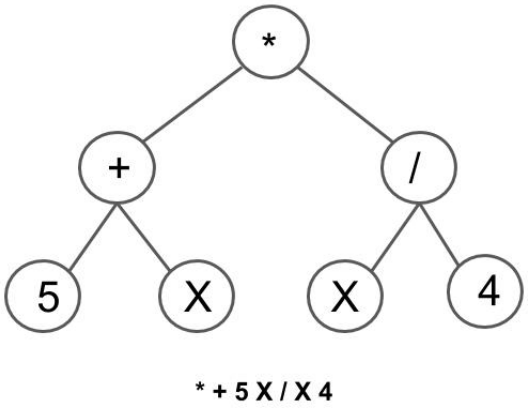


**Figura 1. Exemplo de árvore que pertence a um indivíduo**

## 2.3. Fitness

O *fitness* calculado para cada indivíduo busca encontrar o melhor ajuste tentando minimizando a subtração do resultado encontrado pela árvore com a saída original. A equação 1 foi utilizada para a minimização do nosso objetivo.

$$f(Ind) = Valor\{Ind, x\} - y \tag{1}$$

### 2.3.1. Crossover e Mutação

As operações de cruzamento e mutação são utilizadas para a evolução dos indivíduos e manutenção da diversidade da população. Neste trabalho foi utilizado o cruzamento de um ponto, onde dois pontos são aleatoriamente escolhidos para cortar os respectivos pais e então gerar um filho com as partes que restaram de seus pais. Na Figura 2 podem ver como é realizado o processo de cruzamento no algoritmo implementado.

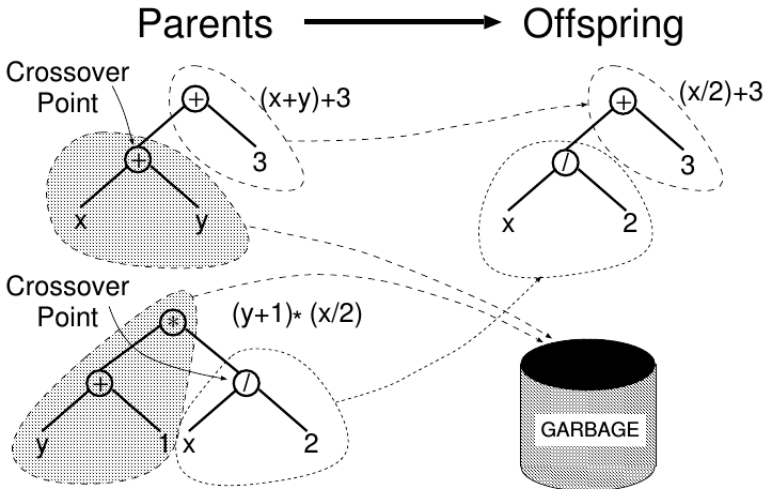


**Figura 2. Exemplo de cruzamento utilizado por [Poli et al. 2008]**

O processo de mutação é bastante similar ao de cruzamento, entretanto não teremos pais retirados da população. De acordo com a taxa de cruzamento, são escolhidos filhos que passarão pelo processo de mutação. Esse processo consiste na geração de uma árvore aleatória e na sua junção com o filho selecionado utilizando a operação de cruzamento. A figura 3 ilustra o operador de mutação implementado nesse trabalho.

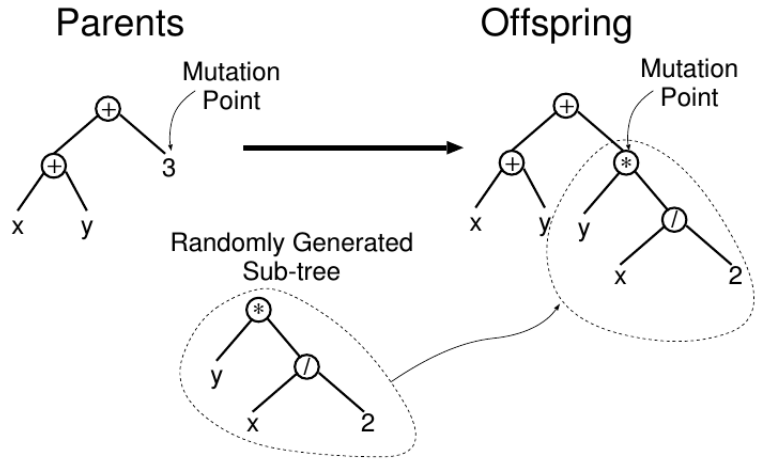


**Figura 3. Exemplo de mutação utilizado por [Poli et al. 2008]**

### 2.3.2. O algoritmo de programação genética

## 3. Implementação

O algoritmo de programação genética utilizado neste trabalho foi implementado na linguagem C++. Para obter uma melhor representação, foram utilizados *structs* para a representação da árvore e do indivíduo,

### 3.1. Estrutura do indivíduo

A *struct* do *induvidual* representa um indivíduo, onde são armazenado um valor do tipo *double* para armazenar a *fitness* e uma *struct* que armazena a raiz da árvore que o representa.

```
struct Individual
{
        struct Tree *tree;
        double fitness;
};
```

Para as operações com as árvores do indivíduo, foi implementada a classe *"tree"*, onde essa possui a *tree* e o *node* que são as representações para os nods que compõem a árvore do indivíduo. Essa classe é constituída do nodo raiz que aponta para os próximos nodos da árvore e um contador de nodo para que possamos sortear o *id* de algum nodos para fazer o cruzamento.

```
struct Tree
{
        struct ExpNode *root;
        unsigned countNodes;
};
```

A *struct ExpNode* que pertence ao arquivo *"tree"* armazena o conteúdo dos nodos que formam a árvore, sendo eles: o *id*, que corresponde a identificação de um nodos; o level, que representa a profundidade em que nodo se encontra; o *kind*, que é o tipo do nodo da árvore. Nessa implementação ele pode assumir três tipo: *VAR*,*NUMBER* ou *OPERATOR*; os apontador *ExpNode* servem para apontar o endereço dos nodos filhos; o *number* é utilizado para armazenar um número caso o tipo da variável seja *NUMBER*; a variável op é utilizada para armazenar o operador quando o nodo é do tipo *OPERATOR*;

```
struct ExpNode
{
        int kind;
        unsigned level;
        double number;
        char op;
        struct ExpNode *left;
        struct ExpNode *right;
        unsigned id;
};
```

### 3.1.1. Métodos importantes do indivíduo

```
Individual *Individual_create()
Individual *Individual_random(unsigned, unsigned )
void individual_crossover(Individual *, Individual *,
   Individual *)
void individual_mutation(Individual *)
```

1. Individual_create: Cria um indivíduo inicializado com valores pré definidos.
2. Individual_random: Cria um indivíduo aleatório pelo método *Full* ou *Grow* de acordo com uma probabilidade de 0.5.
3. individual_crossover: Cruza dois indivíduos selecionados aleatoriamente da população.

4. individual_mutation: Realiza uma mutação de um indivíduo cruzando ele com outro indivíduo gerado aleatoriamente.

#### 3.1.2. Métodos importantes da árvore

```
double getValue(ExpNode *, double);
ExpNode *create_ExpNode(unsigned );
Tree *create_tree();
void Full(Tree *, unsigned);
void Grow(Tree *, unsigned);
ExpNode *add_child(unsigned , ExpNode *, int &);
```

1. getValue: Obtém o valor da expressão da árvore de um indivíduo.
2. create_ExpNode: Cria um nodo para ser alocado em uma determinada árvore.
3. create_tree: Cria uma árvore para ser alocada a um indivíduo.
4. Full: Cria uma árvore aleatória completa.
5. Grow: Cria uma árvore que cresce aleatoriamente até que seja atingido uma certa profundidade.
6. add_child: Adiciona um novo nodo a uma árvore.

### 3.2. Programaçao genética

## 4. Arquivos, compilação e execução

### 4.1. Arquivos

- **main.cpp:** Arquivo principal que realiza a chamadas dos principais métodos.
- **util.cpp:** Métodos úteis que foram utilizados na implementação do algoritmo.
- **tree.cpp:** Arquivo que contém a implementação e os métodos das *structs ExpNodes* e *tree*.
- **database.cpp:** Arquivo utilizado para a leitura da base de dados.
- **geneticProgramming.cpp:** Arquivo onde se encontra implementação do algoritmo de programação genética, bem como os métodos utilizados por ele.
- **util.h:** Header da classe util
- **tree.h:** Header da classe tree
- **database.h:** Header da database
- **geneticProgramming.h:** Header do arquivo de programação genética.

### 4.2. Compilação

A compilação do programa pode ser feito por meio de um Makefile contido na pasta raiz do trabalho. Outro Makefile que pertence a pasta src é executado, assim que o primeiro Make é acionado. Assim, não é necessário executá-lo uma vez que, sua execução é a partir do que está contido na pasta raiz.

- $tp1 - naturalComputing/make$

### 4.3. Execução

Para que o programa seja executado é necessário o seguinte comando:

- $bin/geneticProgramming[npopulation, ngeneration, depth, input/File]$

#### 4.3.1. Entrada

O arquivo de entrada para o algoritmo consiste em m linhas e n colunas, onde n-1 é a quantidade de variáveis da árvore de expressão e a coluna n é a saída esperado pelo calculo da expressão gerada pelo algoritmo. A quantidade m é o número de pontos que será testado.

#### 4.3.2. Saída

Os resultados das execuções exibem uma série de dados. Para cada geração é impresso:

1. Melhor fitness da geração;
2. Pior fitness da geração;
3. Fitness média da geração;
4. Número de indivíduos melhores que os pais (gerados por crossover);
5. Número de indivíduos piores que os pais (gerados por crossover);
6. Número de indivíduos iguais.

## 5. Experimentos

Os experimentos foram executados em um computador com processador Intel Core I7 com 3,5 GHz, memória DDR3 de 12 GB e sistema operacional Ubuntu versão 14.04.3 LTS.

### 5.1. Metodologia

#### 5.1.1. Experimento 1: Convergêcia da população

Nesse experimento foram testados diferentes variações de parâmetros para o algoritmo de programação genética, e nessa seção será analisado qual o impacto dessas variações na procura de uma solução. O conjunto base escolhido, foi o *ellipse_noise* após ser feito uma sequência de testes com parâmetro *default*, nos quais esse conjunto obteve o pior resultado. Portando, tentaremos melhorar o resultado com as variações desses parâmetros.

**Tabela 1. Parâmetros dos experimento**

| População | Nº gerações | Torneio | Elitismo | Crossover | Mutação |
|---|---|---|---|---|---|
| | | | | | |

#### 5.1.2. Experimento 2: Variação do número de gerações

**Tabela 2. Parâmetros dos experimento**

| População | Nº gerações | Torneio | Elitismo | Crossover | Mutação |
|---|---|---|---|---|---|
| | | | | | |

**Tabela 3. Parâmetros dos experimento**

| População | Nº gerações | Torneio | Elitismo | Crossover | Mutação |
|---|---|---|---|---|---|
| | | | | | |

**Tabela 4. Parâmetros dos experimento**

| População | Nº gerações | Torneio | Elitismo | Crossover | Mutação |
|---|---|---|---|---|---|
| | | | | | |

**Tabela 5. Parâmetros dos experimento**

| População | Nº gerações | Torneio | Elitismo | Crossover | Mutação |
|---|---|---|---|---|---|
| | | | | | |

#### 5.1.3. Experimento 3: Variação das probabilidades

#### 5.1.4. Experimento 4: Torneio

#### 5.1.5. Experimento 5: Elitismo

## 6. Resultados

#### 6.0.6. Experimento 1: Convergência da população

#### 6.0.7. Experimento 2: Tamanho da população e número de gerações

#### 6.0.8. Experimento 3: Probabilidades de Mutação e Crossover

#### 6.0.9. Experimento 4: Torneio

#### 6.0.10. Experimento 5: Elitismo

#### 6.0.11. Resultados Gerais

## 7. Conclusão

## Referências

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd.