Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Regressão Simbólica – uma Modelagem por Programação Genética

Caio Felipe Zanatelli.
caio.zanatellidcc.ufmg.br

6 de Outubro de 2018

Resumo

Este trabalho possui como objetivo a criação de um modelo computacional, utilizando Programação Genética, para resolver o problema de Regressão Simbólica. Os conceitos e detalhes do modelo criado serão abordados nas seções seguintes e, além disso, serão apresentados também os resultados obtidos através de testes em ambiente real.

1 Introdução

De acordo com [1], Programação Genética é uma técnica computacional evolucionária cujo que objetiva a resolução automática de problemas, sem que o usuário precise conhecer ou especificar previamente a forma e/ou estrutura do problema central.

A Programação Genética utiliza como base de seu funcionamento a teoria evolucionária de Darwin. Neste contexto, cada possível solução de um problema é representada como um indivíduo, que pertence a um conjunto maior de indivíduos, chamado de *população*. O primeiro passo do algoritmo, então, é gerar uma população aleatória, a partir da qual o processo evolucionário se constituirá. Em seguida, cada indivíduo da população é avaliado através de uma função de aptidão, de forma a verificar o quão próximo da solução ótima do problema este indivíduo é. Esses indivíduos são evoluídos tomando como base suas aptidões, realizando o mesmo processo várias vezes, construindo assim várias gerações. Por fim, o melhor indivíduo presente na última geração é a solução ótima do problema.

Neste trabalho, propomos uma solução para o problema de Regressão Simbólica através do emprego de Programação Genética. Trata-se de um problema de aprendizado, em que se conhece instâncias de uma função (entrada e saída), mas não a função em si, objetivando então a construção de uma expressão simbólica que melhor se ajusta às amostras fornecidas.

2 Modelagem do Problema

No que tange a modelagem do problema, um ponto de grande relevância é a forma de representação de um indivíduo, tendo em vista que será necessário avaliar as expressões várias vezes, e portanto uma boa representação facilita este processo. Sabendo que, em problemas de Regressão Simbólica, cada indivíduo é uma função do tipo $f : \mathbb{R}^m \rightarrow \mathbb{R}$, optou-se por representá-los através de uma árvore binária, o que torna a construção e análise de funções um simples problema de caminhamento em árvores. Neste contexto, os nós internos representam os operadores matemáticos, enquanto os nós folha representam terminais, isto é, variáveis ou constantes. Vale salientar, entretanto, que Programação Genética não requer necessariamente uma árvore binária. No caso deste problema, isso se aplica pois cada operador possui no máximo dois operandos. A Figura 1 ilustra a representação de um indivíduo cuja função é $f(x, y) = 0.95x + cos(y)$.

Inicialmente, o algoritmo foi implementado de forma a suportar as operações de adição, subtração, multiplicação, divisão, seno, cosseno, exponenciação, raíz quadrada e logarítmo. Entretanto, com base em

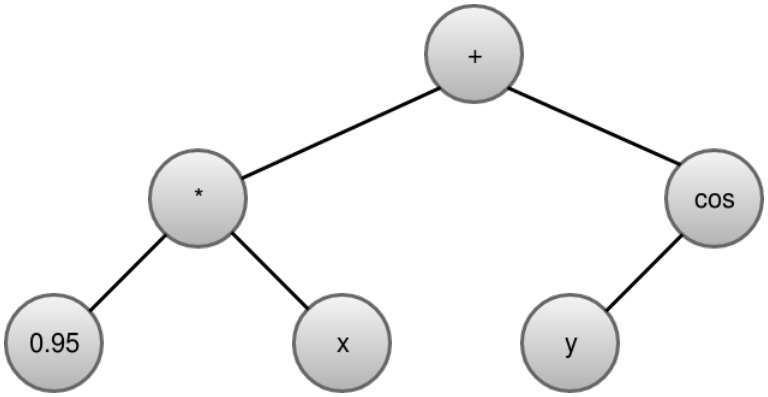

Figura 1: Exemplo de representação de um indivíduo cuja função é $f(x, y) = 0.95x + cos(y)$.

experimentos, observou-se que os seis primeiros operadores eram capazes, por si só, de representar bem várias funções, sem que ocorra *overfitting*, que é um problema em que a função aproxima demais os pontos, e com isso a expressão simbólica encontrada na fase de aprendizado não se torna uma aproximação boa e genérica para novas instâncias. Com isso, os operadores escolhidos incorporam as propriedades de suficiência, fechamento e parcimônia, que são de extrema importância em Programação Genética. Por fim, o conjunto de operadores e terminais adotados são resumidos abaixo:

- **Operadores**:
  - **Binários**:
    - +: operador de adição.
    - **-**: operador de subtração.
    - ***: operador de multiplicação.
    - /: operador de divisão.
  - **Unários**:
    - **sen**: função seno.
    - **cos**: função cosseno.
    - **log**: função logarítmica.
    - **sqrt**: raíz quadrada.
    - **pow**: quadrado de um número.
- **Terminais**:
  - **Constantes**: valores reais gerados aleatoriamente no intervalo $[0, 1]$.
  - **Variáveis**: representação das variáveis da função a ser aproximada. Com isso, o conjunto das variáveis é construído com base na instância do problema, sendo ajustada por parâmetro.

2.1 Cálculo da *Fitness*

Como já abordado, o processo de evolução dos indivíduos é realizado com base em uma função de aptidão (*fitness*). Assim, o critério de avaliação utilizado para medir a qualidade de um indivíduo é a raíz quadrada do erro quadrático médio normalizada (NRMSE), que é calculada da seguinte forma:

$$fitness(Ind) = \sqrt{\frac{\sum_{i=1}^{N}(y_i - EVAL(Ind, x_i))^2}{\sum_{i=1}^{N}(y_i - \overline{Y})^2}},$$

onde N é o número de instâncias fornecidas, Ind é o indivíduo sendo avaliado no momento, EVAL(Ind, x_i) avalia o indivíduo Ind na *i-ésima* instância de X, y_i é a saída esperada para a entrada x_i e $\overline{Y}$ é a média do vetor de saídas Y.

2.2 Geração da População Inicial

Como discutido na introdução, a primeira etapa em Programação Genética é a geração da população inicial. A maneira que essa escolha é realizada é de grande importância para o bom funcionamento do algoritmo, uma vez que deve-se preocupar em gerar uma população diversa o suficiente para que o algoritmo não trave em mínimos/máximos locais logo nas primeiras gerações, melhorando assim a exploração do espaço de busca.

Na literatura, existem diversos métodos para a geração da população inicial. No âmbito deste trabalho, é utilizada uma combinação de dois deles:

(i) ***Grow***: os indivíduos são gerados sem se preocupar com a altura da árvore, sendo a única restrição não ultrapassar aquela configurada como máxima, possibilitando indivíduos com variação considerável de genótipo.

(i) ***Full***: os indivíduos são gerados utilizando a altura máxima configurada para a árvore que representa a expressão simbólica, isto é, todas as folhas estão na mesma profundidade.

Por fim, de forma a garantir a diversidade da população inicial, e com isso possibilitar uma melhor exploração do espaço de busca, o método adotado foi o ***Ramped Half-and-Half***, isto é, uma combinação dos métodos *Grow* e *Full*, sendo metade dos indivíduos gerados com o primeiro e a outra metade com o segundo método.

2.3 Evolução das Gerações

Após a geração da população inicial, o algoritmo entra em uma repetição de rodadas, onde cada rodada é responsável por evoluir a geração corrente, gerando populações cada vez melhores, de forma a convergir a solução construída para a solução ótima do problema. Neste contexto, a evolução dos indivíduos é realizada através de seleção e aplicação de operadores genéticos, tópicos esses que serão abordados nas subseções seguintes. Ainda, de forma a proporcionar uma melhor compreensão acerca do processo evolutivo, a Figura 2 ilustra essas etapas.

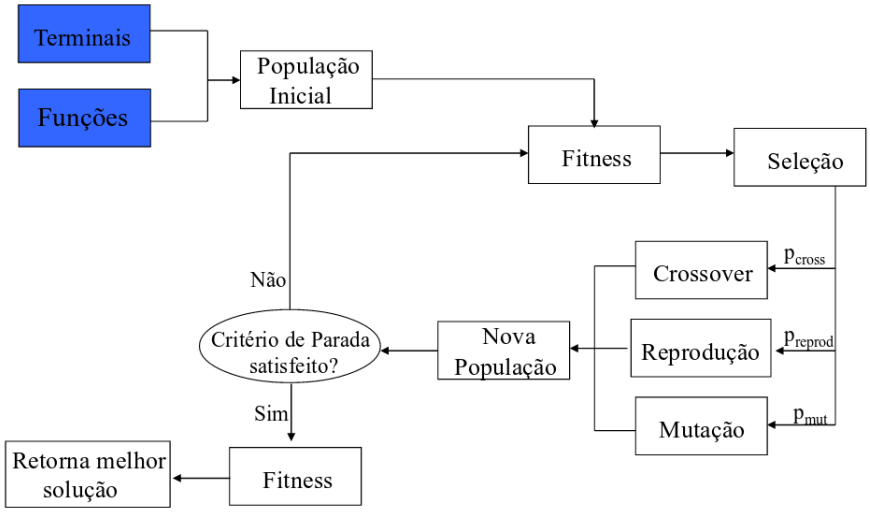

Figura 2: As etapas do arcabouço de Programação Genética.

2.3.1 Seleção

Para que o algoritmo melhore os indivíduos para as próximas gerações, é desejável criar um mecanismo que mantenha as características dos indivíduos que possuem melhor aptidão na população em que se encontram. Para tanto, foi adotada a seleção por **Torneio**.

A seleção implementada, portanto, escolhe aleatoriamente k indivíduos da população e retorna aquele com a maior *fitness*, onde o k é um parâmetro configurado pelo usuário que dita o tamanho do torneio, isto é, a quantidade de indivíduos que serão selecionados para se confrontarem no processo de seleção.

Por fim, este trabalho também implementa **Operadores Elitistas**, de forma a garantir que o indivíduo que será transferido para a próxima população seja sempre o melhor da seleção. O funcionamento básico desses operadores é descrito abaixo:

1. A operação genética é executada.
2. Se o filho gerado for melhor que ambos os pais, ele é adicionado à próxima geração. Caso contrário, o melhor pai é adicionado à próxima geração.

Vale ressaltar, neste contexto, que os operadores elitistas são a opção padrão, entretanto é possível configurar o algoritmo para não utilizá-los, o que pode ser feito através da passagem de parâmetros no momento da execução do algoritmo.

2.3.2 Operadores Genéticos

Após a seleção dos indivíduos, são aplicados os operadores genéticos, com produzir novos indivíduos a partir de operações com o genótipo daqueles selecionados, que em tese se destacaram na população, o que é fundamental para que a solução possa convergir. São três os operadores genéticos, e cada um possui uma função específica para a evolução das gerações:

(i) **Cruzamento**: ocorre com probabilidade pré-estabelecida. Dois indivíduos, ditos pais, são selecionados via Torneio. Para cada pai, é escolhida uma subárvore aleatória de seu genótipo, e troca-se as subárvores entre os pais, gerando assim dois novos indivíduos filhos. Observe que os pais não são alterados neste processo.

(ii) **Mutação**: ocorre com probabilidade pré-estabelecida. Um único indivíduo é selecionado via Torneio, e uma subárvore de seu genótipo, escolhida aleatoriamente, é substituída por outra subárvore também gerada aleatoriamente.

(iii) **Reprodução**: Um indivíduo selecionado é repassado para a próxima geração, levando em consideração se a opção de operadores elitistas está ativa ou não.

3 Implementação

Uma vez discutidos os conceitos e a modelagem realizada, esta seção tem como objetivo apresentar as decisões de implementação e a estrutura do código, além de fornecer instruções para a sua execução.

3.1 Decisões de Implementação

Este trabalho foi inteiramente desenvolvido utilizando a linguagem *Python 2*. Abaixo estão as principais decisões de implementação tomadas neste projeto.

- O conjunto de operadores utilizados foram *soma*, *subtração*, *multiplicação*, *divisão*, *seno* e *cosseno*, uma vez que apresentaram melhores resultados se comparado com um conjunto mais extenso de operadores, favorecendo assim a suficiência, o fechamento e a parcimônia, propriedades desejáveis em Programação Genética. Entretanto, o código foi modularizado de tal forma que é possível realizar a extensão do conjunto de operadores, caso seja desejado;

- Para os terminais constantes, foram gerados aleatoriamente valores reais entre 0 e 1. Foi adotado este intervalo uma vez que se mostraram suficientes para a convergência da solução, tendo em vista que constantes muito altas podem afastar muito a solução obtida da ótima;
- Para os operadores unários, como *seno* e *cosseno*, existe apenas um operando. Neste contexto, foi adotado como padrão utilizar o nó à esquerda como o parâmetro da função trigonométrica e deixar o nó à direita vazio, continuando a construção normal da árvore;
- Para o operador de divisão, foi estabelecido um critério de divisão segura, isto é, caso ocorra uma divisão por zero, o valor retornado é zero;
- A altura máxima da árvore que representa um indivíduo foi configurada como 7, e não é variada;
- Para o cálculo da *fitness*, caso um indivíduo exceda a profundidade máxima da árvore, em casos de *crossover* ou mutação, foi aplicada uma penalidade à sua *fitness*, sendo atribuído o maior valor inteiro representável em *Python*.

3.2 Estrutura do Projeto

O código foi modularizado de forma a proporcionar futuras extensões, tornando o algoritmo de Programação Genética algo genérico para outros problemas, sendo necessário alterar apenas a representação do indivíduo e o cálculo da *fitness*. A estrutura adotata é apresentada a seguir.

- **chromossome.py**: define a classe *Chromossome*, responsável por representar a árvore de um indivíduo (genótipo). Fornece métodos relativos ao modelo em árvore, tais como a geração de subárvores aleatórias, caminhamento em árvore, procura e substituição de um nó, etc.
- **individual.py**: define a classe *Individual*, responsável por representar um indivíduo de uma população, o qual é composto de seu genótipo (*Chromossome*) e sua *fitness* associada. Fornece métodos para o cálculo da *fitness* dada uma instância do problema e para a aplicação de mutação em seu genótipo.
- **population.py**: define a classe *Population*, responsável por representar uma população de indivíduos. Fornece métodos para adição e pesquisa na lista de indivíduos e métodos para a geração da população inicial, a qual foi devidamente modularizada de forma a possibilitar a inclusão de outras formas de geração, sendo o *Ramped Half-and-Half* a padronizada neste projeto. Além disso, fornece também um método para o cálculo da *fitness* de todos os indivíduos da população, o qual utiliza os métodos definidos pela classe *Individual*.
- **statistics.py**: define a classe *Statistics*, responsável por armazenar e prover métodos para o cálculo de estatísticas referentes à cada geração.
- **utils.py**: define a class *Utils*, responsável por prover funções variadas que são comuns nos demais módulos, tais como: verificar se um terminal é uma variável ou uma constante, gerar uma probabilidade aleatória, gerar um valor inteiro aleatório, gerar um valor real aleatório, etc.
- **ioutils.py**: define a classe *IOUtils*, responsável por fornecer funções relativas à manipulação de arquivos, como escrita e leitura de arquivos CSV.
- **genprog.py**: define a classe *GeneticProgramming*, responsável por representar a Programação Genética em si. Coordena todo o fluxo do algoritmo: geração da população inicial e evolução das gerações através de seleção e da aplicação de operadores genéticos. Além do método para a fase de treinamento, fornece também um método para a fase de testes, a qual avalia uma instância do problema a partir da solução construída na fase de aprendizado.
- **syregression.py**: programa principal. Trata-se do módulo executado pelo usuário, sendo responsável por realizar o *parsing* dos parâmetros e por efetuar as chamadas para dar início às fases de treinamento e de testes, através de uma instância da classe *GeneticProgramming*.

3.3 Execução e Parâmetros

De forma a tornar o projeto adaptável, optou-se por utilizar parâmetros por linha de comando para definir os pontos chave do modelo utilizado. A Tabela 1 apresenta a lista de parâmetros permitidos.

Tabela 1: Lista de parâmetros.

Parâmetro	**Descrição**	**Valor Padrão**
--h, --help	Exibe menu de ajuda	—
--train	Caminho do arquivo de treinamento	—
--test	Caminho do arquivo de teste	—
--out	Caminho do arquivo de saída	—
--pop-size	Tamanho da população	30
--cross-prob	Probabilidade de cruzamento	0.9
--mut-prob	Probabilidade de mutação	0.05
--max-depth	Altura máxima da árvore	7
--seed	Semente para a geração de números aleatórios	1
--nvariables	Número de variáveis para construir a função	2
--ngen	Número de gerações	30
--tour-size	Tamanho k para a seleção por Torneio	2
--elitism	Ativa a utilização de operadores de elitismo	*True*

Como é possível observar na Tabela 1, vários parâmetros são opcionais. Portanto, o código para a execução do programa é o seguinte, onde parâmetros entre colchetes indicam que são opcionais:

```
$ python syregression [-h] --train TRAIN --test TEST --out OUT
                      [--pop-size POP_SIZE] [--cross-prob CROSS_PROB]
                      [--mut-prob MUT_PROB] [--max-depth MAX_DEPTH]
                      [--seed SEED] [--nvariables NVARIABLES] [--ngen NGEN]
                      [--tour-size TOUR_SIZE] [--elitism ELITISM]
```

3.4 Entrada e Saída

Os arquivos de entrada são dados em formato CSV e as entradas são dadas em ponto flutuante. Considerando n o número entradas uma linha do arquivo, os $n-1$ primeiros valores correspondem às entradas para as variáveis da função: $x_1, x_2, ..., x_{n-1}$, e a *n-ésima* entrada corresponde à saída y da função quando aplicadas as variáveis $x_1, x_2, ..., x_{n-1}$, isto é, $y = f(x_1, x_2, ..., x_{n-1})$.

Em relação à saída, o código produz dois arquivos: *out.txt* e *out.txt.csv*, onde *out* é o arquivo passado pelo parâmetro *--out*. O primeiro arquivo armazena um *log* formatado com as informações pertinentes a todas as gerações da Programação Genética, para consulta humana, enquanto o segundo arquivo gera o mesmo resultado do *log* anterior, porém em formato CSV. O último arquivo é utilizado como entrada para *scripts* criados para a análise experimental. Abaixo é apresentado um trecho de do arquivo de saída formatado e do *log* CSV.

```
[+] Generation:       29
[+] Average Fitness: 1.19581781311
[+] Best Fitness:    0.970392938911
[+] Worst Fitness:   3.11969813851
[+] Num. of repeated invididuals: 12
[+] Num. of crossover-generated indivs. that are better than their parents: 20
[+] Num. of crossover-generated indivs. that are worse than their parents:  8
```

```
29,1.19581781311,0.970392938911,3.11969813851,12,20,8
```

4 Análise Experimental

Tendo em vista que algoritmos evolucionários dependem fortemente de seus parâmetros, esta seção tem como objetivo apresentar os experimentos realizados para a validação de seu funcionamento, bem como o tunelamento dos parâmetros associados. Todos os experimentos apresentados a seguir foram realizados em uma máquina *Intel Core* i5-3470 de 3.20GHz, com quatro núcleos; 16GB de memória RAM e sistema operacional CentOS 7. Vale ressaltar, ainda, que, por se tratar de um processo estocástico, cada experimento foi realizado 30 (trinta) vezes, tomando a média dos valores obtidos como padrão para efeitos de comparação.

Por fim, vale ressaltar que os operadores matemáticos escolhidos para a base **synth1** foram *+, -, *, /, log, pow, sqrt*, para a base **synth2** foram *+, -, *, /, sin, cos*.

4.1 Experimento 1 – Definindo Parâmetros Iniciais

O primeiro experimento realizado teve como objetivo definir dois parâmetros iniciais, são eles: o tamanho da população e o número de gerações. Para tanto, foi utilizada a base de dados ***synth1***, variando ambos os parâmetros e observando o comportamento e convergência da solução ótima encontrada. Para o teste em questão, os parâmetros supracitados foram variados em 50, 100 e 500, de forma a encontrar a melhor configuração para dar prosseguimento ao ajuste dos demais parâmetros. As Figuras 3–11 ilustram os resultados obtidos, apresentando a melhor solução encontrada em cada uma das combinações, para cada geração do algoritmo. Através dos gráficos, portanto, pode-se comprovar que, de fato, o algoritmo converge para a solução ótima.

Ainda, é possível observar que, como esperado, ao aumentar o tamanho da população, a solução tende a melhorar. Isso ocorre pois ao aumentar o tamanho da população, aumentamos o espaço de busca, e com isso obtém-se mais diversidade, sendo 500 o tamanho da população que gerou melhores resultados. Além disso, ao aumentar o número de gerações, percebe-se uma melhora significativa na solução, uma vez que tem-se mais rodadas para que a solução possa convergir. Entretanto, ao se aproximar da geração 100, a melhor solução não se altera muito, e portanto o custo computacional para se utilizar mais gerações é inviável. Neste contexto, constata-se que estabelecer o tamanho da população e o número de gerações como 500 e 100, respectivamente, é uma boa escolha para esta base.

Por fim, a Tabela 2 sintetiza os resultados obtidos para esses testes.

Tabela 2: Resumo dos experimentos de variação da população e de gerações.

Tamanho da População	**Número de Gerações**	**Erro Mínimo**
50	50	0.0796073153686
50	100	0.0814969638152
50	500	0.0246072851819
100	50	0.0972187689708
100	100	0.0812320585935
100	500	0.0251593676529
500	50	0.0138319040293
500	100	0.0137518560373
500	500	0.00106480251181

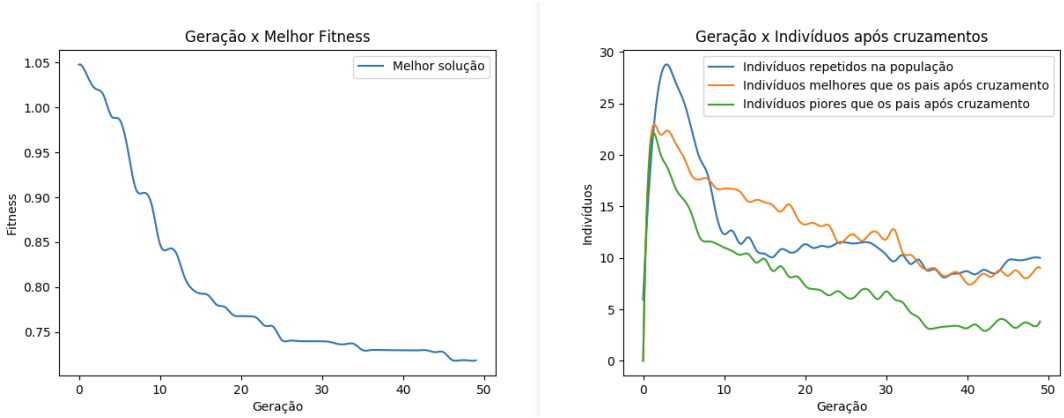

Figura 3: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 50, número de gerações 50, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

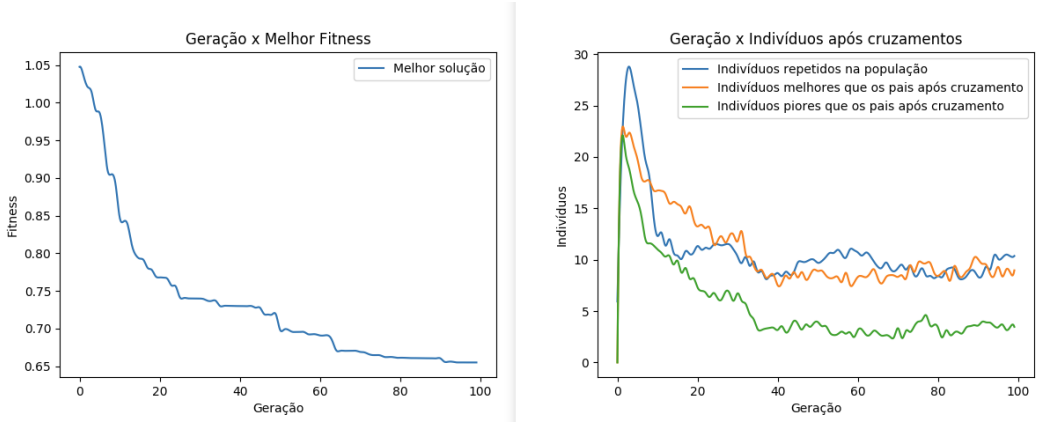

Figura 4: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 50, número de gerações 100, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

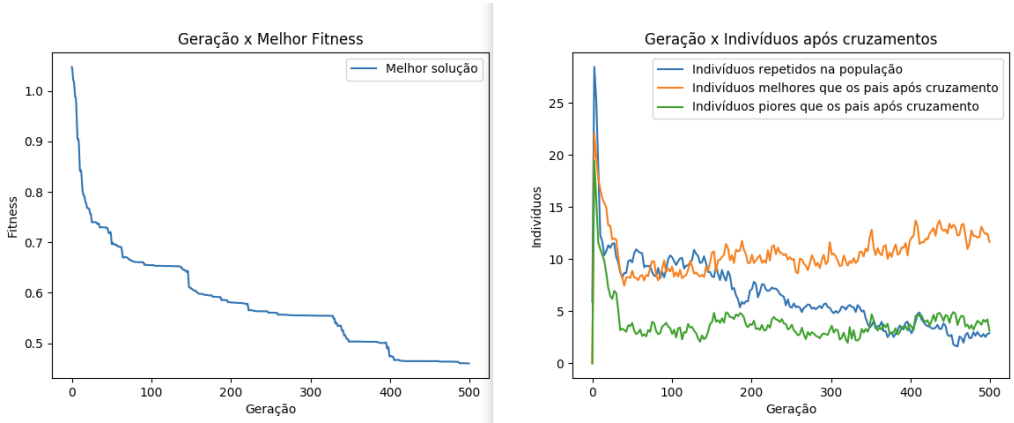

Figura 5: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 50, número de gerações 500, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

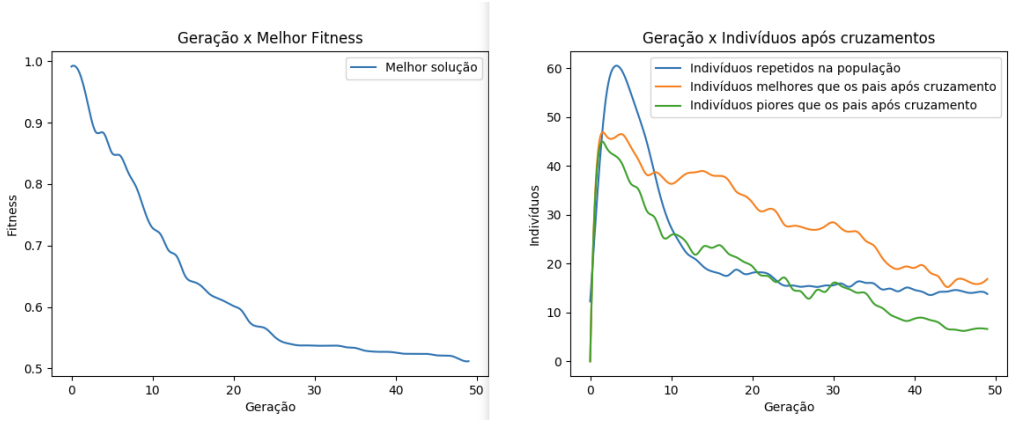

Figura 6: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 100, número de gerações 50, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

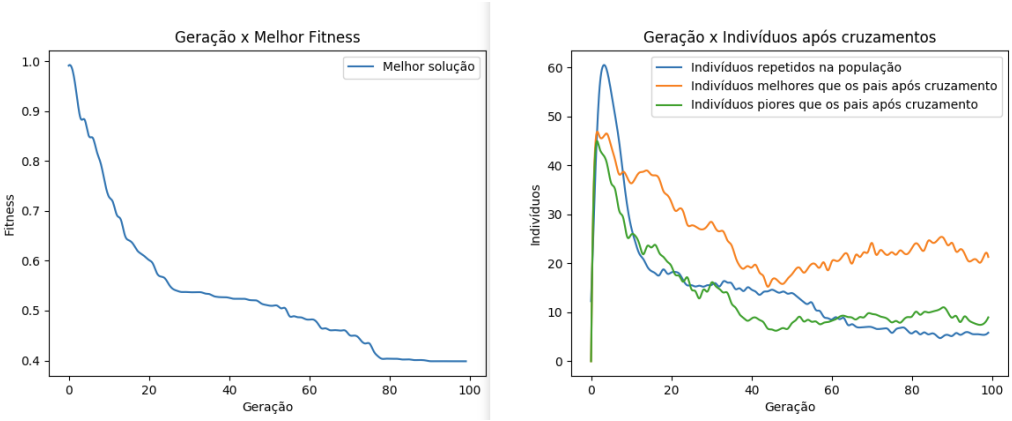

Figura 7: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 100, número de gerações 100, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

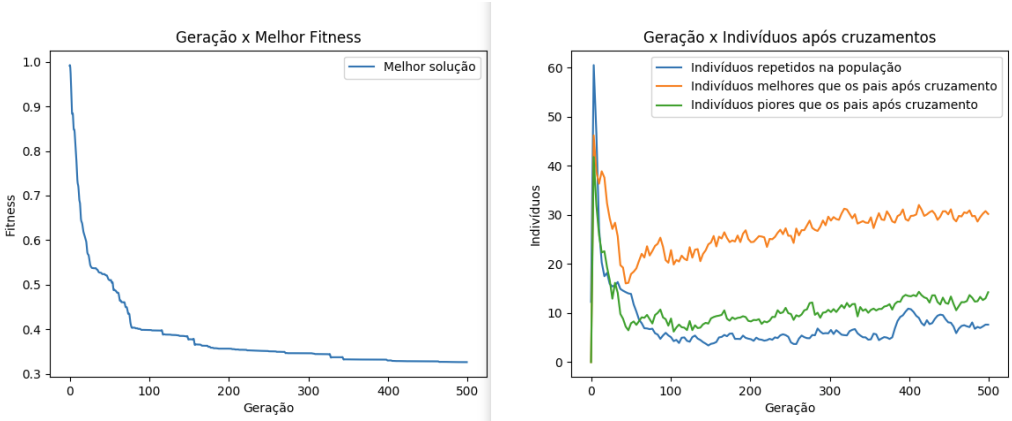

Figura 8: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 100, número de gerações 500, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

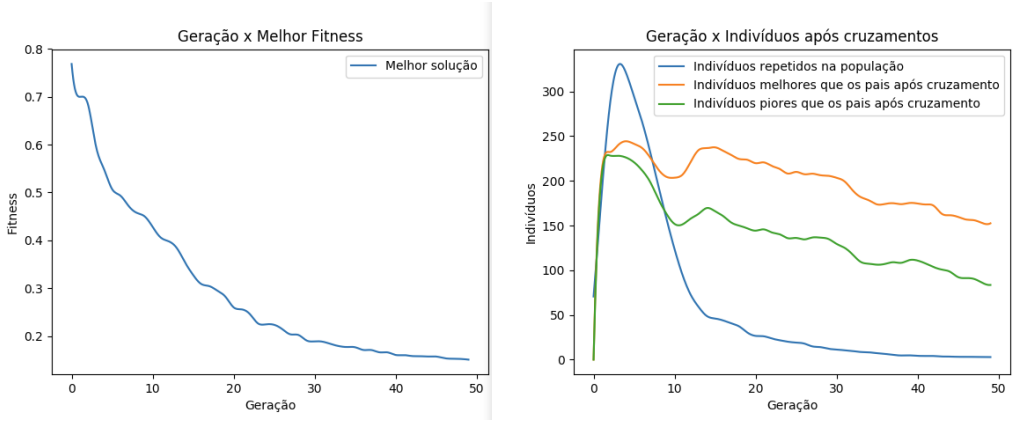

Figura 9: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 50, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

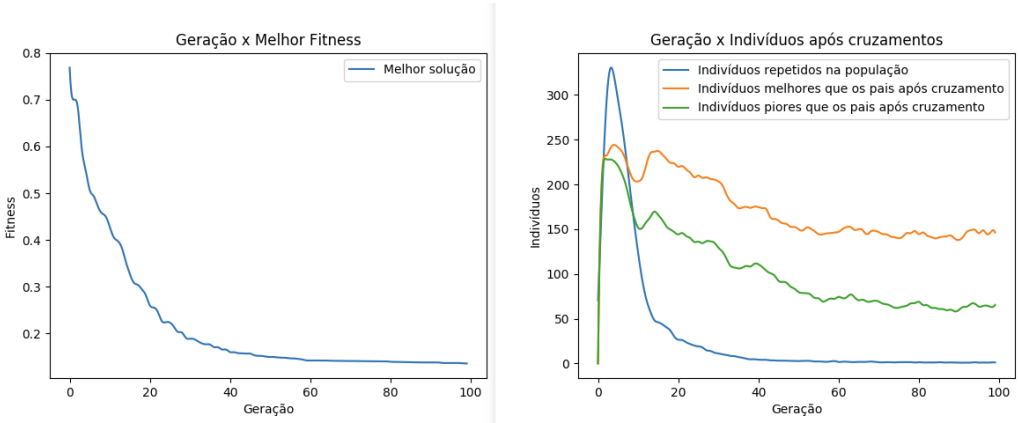

Figura 10: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 100, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

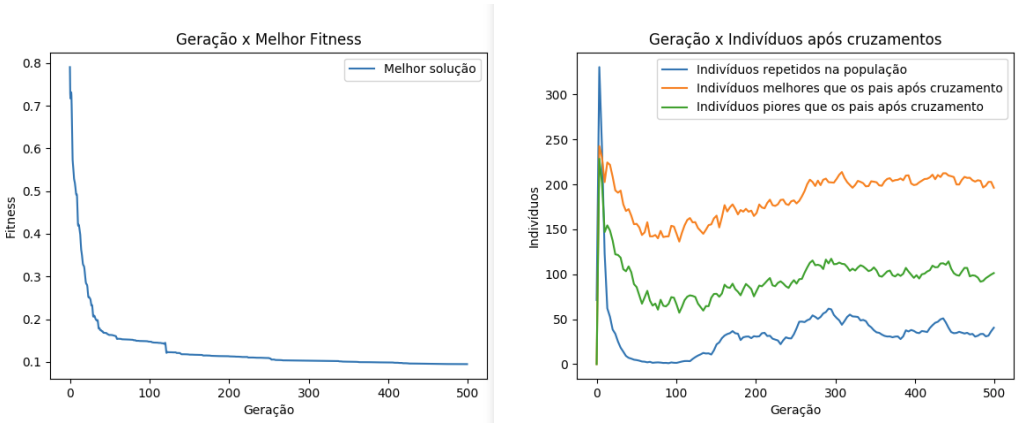

Figura 11: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 500, $K_{tour} = 2$, $Pc = 0.9$ e $Pm = 0.05$.

4.2 Experimento 2 – Avaliando Cruzamento e Mutação

O segundo experimento realizado consiste em analisar o impacto das probabilidades de cruzamento e de mutação sobre o algoritmo. Foram testados dois cenários, mantendo um dos parâmetros baixo e o outro alto. Para tanto, utilizou-se os parâmetros obtidos no experimento anterior, isto é, população de tamanho 500 e número de gerações igual a 100.

Para a mutação, foi avaliado o comportamento do algoritmo quando a probabilidade em questão é elevada. Para tanto, foram testadas os valores 0.20, 0.25 e 0.30 para a probabilidade de mutação, mantendo a probabilidade de cruzamento em 0.6. A 3

Tabela 3: Resumo dos experimentos de variação da probabilidade de mutação.

Probabilidade de Cruzamento	**Probabilidade de Mutação**	**Erro Mínimo**
0.60	0.20	0.01856734604430
0.60	0.25	0.00115117344817
0.60	0.30	0.02617422631890

Como é possível observar na Tabela 3, ao fazer $P_m = 0.25$ e $P_c = 0.60$, obteve-se a redução do erro mínimo em uma ordem. isto é, o erro mínimo para a base **synth1** foi reduzido de 0.0137518560373 para 0.00115117344817, com a nova configuração dos parâmetros. De fato, se a taxa de cruzamento é muito alta nas gerações iniciais, pode ocorrer um problema de pressão seletiva, e com isso levar a uma melhor solução local. Nesta situação, a velocidade de convergência é bem maior, entretanto o resultado pode não ser o desejado, e portanto a operação de mutação é importante para que haja diversidade na população, de forma a evitar a pressão seletiva. Além disso, observando as Figuras 12, 13 e 14, percebe-se, também, que a quantidade de indivíduos piores que os pais diminui na medida em que a taxa de mutação aumenta.

Por fim, pode-se dizer que os parâmetros referentes ao tamanho da população e ao número de gerações ainda são adequados, uma vez que a convergência é estável (Figura 13). Neste contexto, de acordo com os resultados obtidos, optou-se por estabelecer os parâmetros de mutação como 0.25 e o de cruzamento como 0.60, uma vez que esta configuração apresentou melhores resultados para a base escolhida. Entretanto, optou-se, também, por aumentar o número de gerações para 150, com o intuito de tentar reduzir ainda mais o erro mínimo obtido.

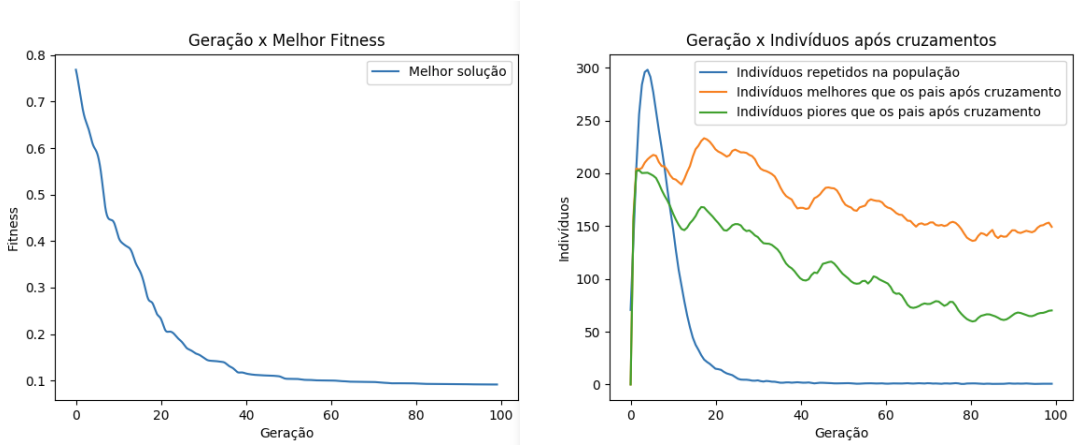

Figura 12: Experimento 2 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 100, $K_{tour} = 2$, $Pc = 0.60$ e $Pm = 0.20$.

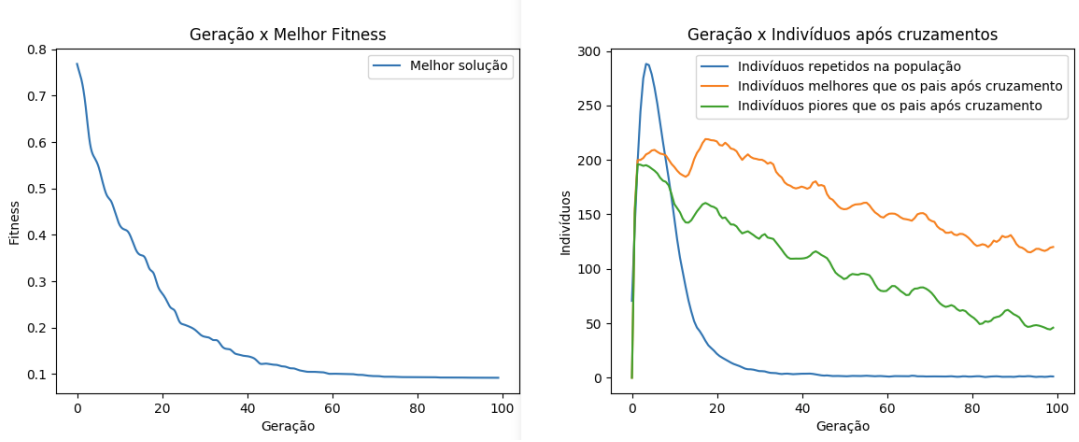

Figura 13: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 100, $K_{tour} = 2$, $Pc = 0.6$ e $Pm = 0.25$.

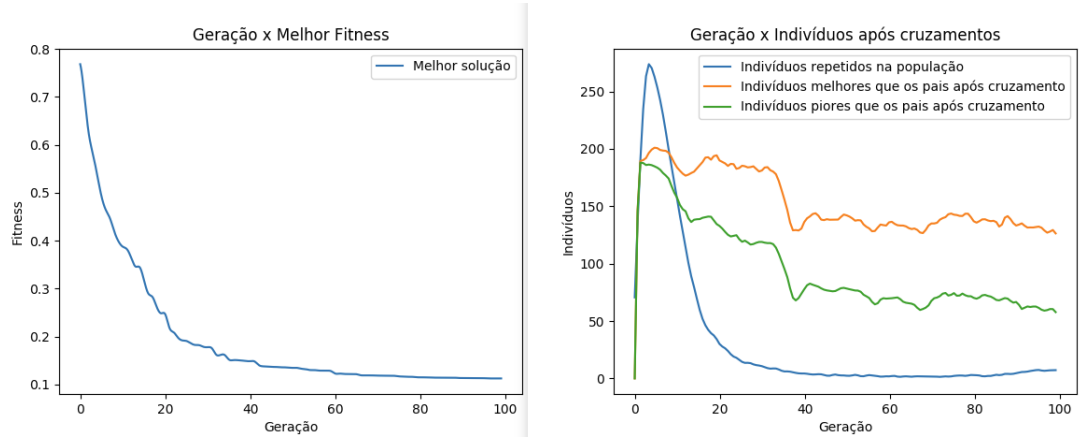

Figura 14: Experimento 1 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 100, $K_{tour} = 2$, $Pc = 0.60$ e $Pm = 0.30$.

4.3 Experimento 3 – Avaliando o Tamanho do Torneio

O terceiro experimento realizado refere-se à escolha do tamanho do torneio. Tendo em vista que, até aqui, os testes foram realizados com $K_{tour} = 2$, a Figura 15 ilustra o comportamento do algoritmo ao ajustar tal parâmetro para 5. O resultado indica uma melhoria significativa na convergência da solução, além do fato que o número de indivíduos melhores que os pais ao longo das gerações melhorou, em contraste com a diminuição dos indivíduos piores que os pais, comprovando que o aumento do tamanho do torneio favorece a escolha de indivíduos melhores, aumentando, assim, a qualidade das soluções obtidas ao longo do tempo. Por fim, também foi testado o tamanho de torneio igual a 7, entretanto as soluções obtidas foram piores do que aquelas obtidas com a configuração de tamanho 5, pois a solução convergia muito rápido e estagnava em um mínimo local. Portanto, constata-se que a seleção do tamanho do torneio é fundamental para o bom funcionamento do algoritmo, tendo que ser grande o suficiente para incluir bons indivíduos, mas não muito grande, pois é importante para manter a diversidade. Neste contexto, é importante ressaltar que o erro mínimo obtido, mais uma vez, melhorou em uma ordem, passando de 0.00115117344817 (Experimento 2) para 0.000386599796797, com o aumento do parâmetro K_{tour} para 5.

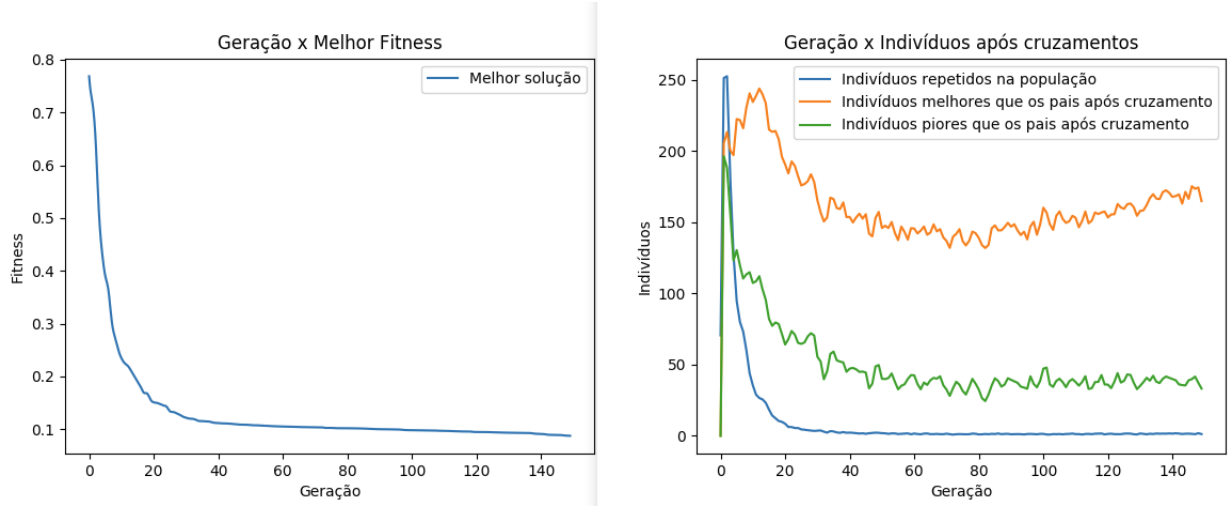

Figura 15: Experimento 3 – Geração x Melhor *fitness* para a base **synth1**. Configuração: tamanho da população 500, número de gerações 150, $K_{tour} = 5$, $Pc = 0.60$ e $Pm = 0.25$.

4.4 Experimento 4 – Avaliando o Impacto dos Operadores Elitistas

Os operadores elitistas se mostraram importantes nos experimentos anteriores, contribuindo para a convergência mais rápida das soluções. Isso ocorre porque os indivíduos mais aptos são reproduzidos para a próxima geração. Além disso, é importante ressaltar, também, que solução pode piorar, em relação a uma geração anterior, quando operadores elistitas não são utilizados. Com isso, o elitismo se mostra importante para garantir que a solução sempre melhore ou se mantenha constante.

4.5 Avaliando de *Bloating*

O *bloating* é um problema recorrente em Programação Genética, uma vez que, ao utilizar muitas gerações, os indivíduos tendem a crescer. Quando crescem incontrolavelmente, o tamanho máximo da árvore é alcançado, e isso causa *overfitting* no algoritmo, além de muita memória ser utilizada, e com isso aumentar gradativamente o tempo de processamento.

Neste trabalho, a medição de *bloating* não é uma tarefa árdua, uma vez que foram criadas funções para a contagem da profundidade das árvores dos indivíduos, e com isso é possível estabelecer métricas para verificar se ocorreu *bloating* com base nos tamanhos dos cromossomos. Além disso, foram implementadas duas formas de previnir *bloating*: penalização da *fitness* de um indivíduo caso ele ultrapasse o tamanho máximo permitido, e a correção de uma árvore, através de poda, caso ela esteja maior do que deveria.

4.6 Experimento 5 – Avaliação da Base de Dados *synth2*

Para a base **synth2**, utilizou-se os mesmos parâmetros obtidos nas subseções anteriores, de forma a avaliar o desempenho do algoritmo nessa base. A Tabela 4 resume os parâmetros utilizados.

Percebe-se, pelos gráficos da Figura 16, que, os parâmetros ótimos encontrados para a base **synth1** não são mais apropriados para a base **synth2**. A curva, de fato, converge para uma eventual solução ótima, entretanto, a velocidade de convergência é muito baixa, fazendo com que as soluções sejam pouco melhoradas ao longo das gerações. Isso pode ter ocorrido pelo tamanho da população inicial ter sido razoavelmente alto (500), e com isso o espaço de busca se tornou vasto demais para esta base em específico, o que não ocorreu para a base anterior. Portanto, uma tentativa de melhorar os parâmetros seria aumentar o número de gerações, de forma a possibilitar mais tempo para a convergência das soluções, mas, sobretudo, aumentar a probabilidade de mutação e o tamanho do torneio, de forma a garantir que indivíduos aptos tenham mais chance de ser escolhidos.

Tabela 4: Resumo dos experimentos de variação da probabilidade de mutação.

Parâmetro	**Valor**
Tamanho da população	500
Número de gerações	100
Probabilidade de cruzamento	0.60
Probabilidade de mutação	0.24
Tamanho do torneio	5
Elitismo	Sim

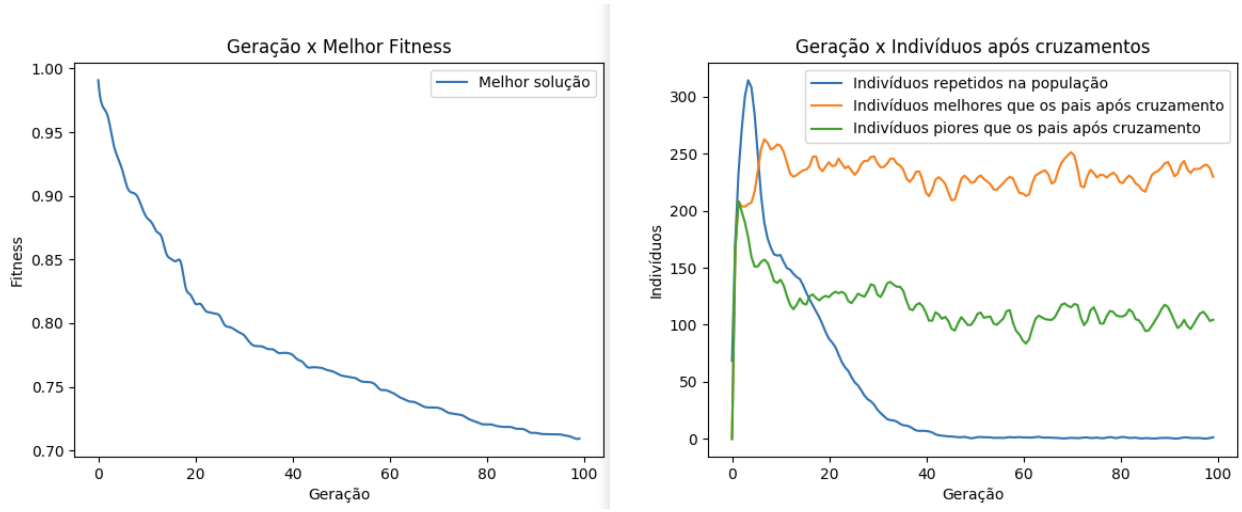

Figura 16: Experimento 5 – Geração x Melhor *fitness* para a base **synth2**. Configuração: tamanho da população 500, número de gerações 100, $K_{tour} = 5$, $Pc = 0.60$ e $Pm = 0.25$.

4.7 Experimento 5 – Avaliação da Base de Dados *concrete*

Como já dito anteriormente, para a base **concrete** foi utilizado o seguinte conjunto de parâmetros: *+, -, *, /, pow, log, sqrt*. A análise de resultados para a base em questão foi a mais árdua, uma vez que não conhecemos a forma do gráfico da função que buscamos, e com isso definir os operadores, por si só, já foi difícil. Foram testados vários conjuntos, sendo o supracitado aquele que melhor se aproximou dos resultados esperados.

Além disso, um ponto de dificuldade para esta avaliação está centrada no tamanho das entradas, uma vez que são oito variáveis e a base é maior. Neste sentido, utilizar populações de tamanho elevado, como aquele utilizado para as bases **synth1** e **synth2** (500), se torna inviável neste cenário. Aliado a isso, o número de gerações também teve de ser reduzido, pois a complexidade e o tempo de processamento aumenta consideravelmente.

Optou-se por utilizar 200 como tamanho inicial da população, pois, para valores maiores que isso, o tempo de processamento era muito elevado, e para valores menores o algoritmo estagnava em ótimos locais. O número de gerações, por outro lado, foi fixado em 50. Os testes realizados mostraram que esse número deveria ser maior, uma vez que o algoritmo demora a convergir com as funções utilizadas para esta base. Isso se deve ao fato de que o tamanho da população inicial influencia na escolha do número de gerações, pois, se configurarmos um valor razoável, como foi o caso, deve haver um número de gerações suficiente para que haja a convergência da solução, dado que, neste cenário, há uma exploração maior do espaço de busca.

Em relação às probabilidades de mutação e cruzamento, após vários testes, foi decido por utilizar $Pc = 0.60$ e $Pm = 0.25$. Em relação aos testes iniciais, em que as probabilidades de cruzamento e de mutação foram fixadas em 0.90 e 0.05, respectivamente, houve uma melhora significativa nos resultados. Como era

esperado, como maior taxa de mutação, aumentamos o espaço de busca, evitando, assim, que o algoritmo trave em ótimos locais. Entretanto, a escolha desses parâmetros foi bem difícil, e com certeza não é a ótima, uma vez que o espaço de busca parece estar amplo demais, e com isso a solução final converge devagar demais.

Além disso, o tamanho do torneio foi testado com valores 2, 3, 4 e 5. O tamanho adequado para os parâmetros escolhidos anteriormente foi 3. Esta escolha foi tomada porque experimentos mostraram que, com tamanho 2, a convergência era muito lenta, e com valores maiores que 3, a convergência era muito rápida, porém para soluções ótimas locais. Com tamanho 3, entretanto, o espaço de busca foi bem explorado e a solução convergiu bem.

Por fim, nota-se que os parâmetros escolhidos são adequados para a base **concrete**, entretanto podem ser melhorados. Para este trabalho, não tivemos tempo suficiente para realizar todas as combinações de parâmetros como fizemos para a base **synth1** e, com isso, o erro mínimo obtido aqui foi mais alto do que o esperado. Entretanto, foi possível observar e analisar os impactos de cada parâmetro, como indica as análises realizadas nesta subseção. Principalmente, ficou claro, também, o quanto Programação Genética depende de seus parâmetros, mas, sobretudo, sua dependência forte com a base de dados utilizada. Os resultados obtidos são ilustrados pelo gráfico da Figura 17.

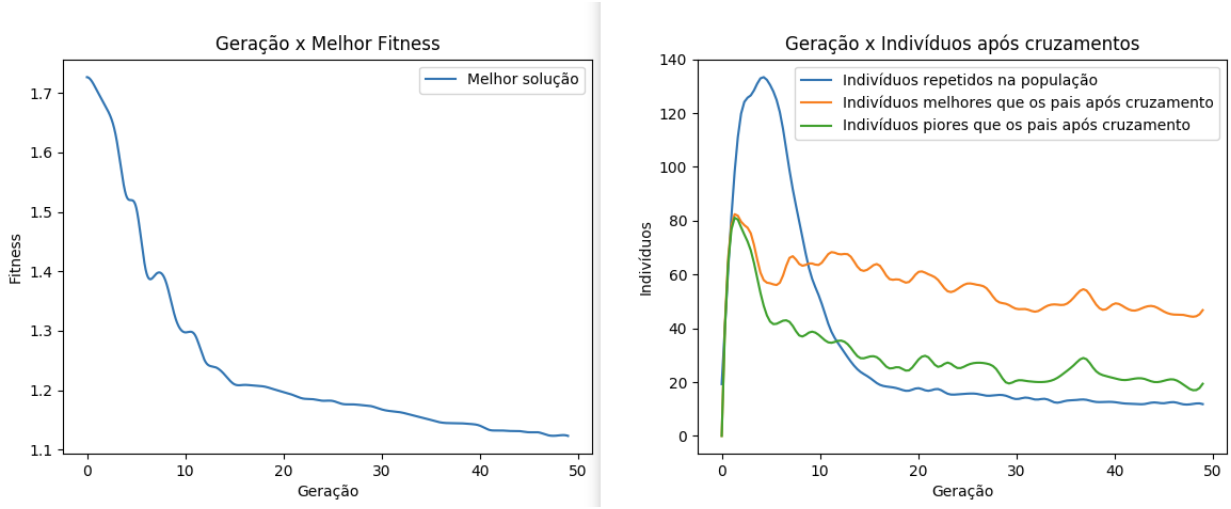

Figura 17: Experimento 5 – Geração x Melhor *fitness* para a base **concrete**. Configuração: tamanho da população 200, número de gerações 50, $K_{tour} = 3$, $Pc = 0.60$ e $Pm = 0.25$.

5 Conclusão

Este trabalho apresentou uma modelagem para o problema de Regressão Simbólica através de Programação Genética, que é uma técnica bioinspirada poderosa que permite a construção da solução de um problema sem saber exatamente sua forma. Neste contexto, este projeto foi de grande importância para esclarecer e fixar conceitos vistos em sala de aula. Além disso, Programação Genética é uma modelagem genérica e, com isso, certamente o algoritmo desenvolvido poderá ser reutilizado para outros fins, uma vez que todos os módulos foram desenvolvidos de forma a reduzir o acoplamento do projeto como um todo.

Ainda, através desse trabalho foi possível praticar a modelagem de problemas, algo que muitas vezes é abstrato demais e, portanto, foi de grande relevância para compreender melhor o funcionamento de Programação Genética. Entretanto, a maior dificuldade encontrada está relacionada à execução dos testes, uma vez que o algoritmo requer um custo computacional considerável, dependendo fortemente do tamanho da entrada e dos parâmetros configurados. Portanto, realizar a implementação e todos os testes para o tunelamento adequado de parâmetros foi um desafio neste trabalho, mas permitiu a visualização da importância de do impacto de cada um deles.

Referências

[1] Poli, R.; Langdon, W. B.; and McPhee, N. F.. *A field guide to genetic programming*. LuLu, 1st edition, 2008.