Computação Natural

Trabalho Prático I - 2024/2

Igor Lacerda Faria da Silva

1. Introdução

O Trabalho Prático I de Computação Natural consistiu na implementação de uma Programação Genética (PG) para uso como função de distância do algoritmo de *clustering* do `scikit-learn`. A métrica usada na avaliação foi o V-Measure Score (Métrica V). Foram utilizados duas bases de dados como referência: o Breast Cancer e o Wine Red. O repositório deste trabalho pode ser encontrado neste link.

A documentação está divida da seguinte maneira: esta breve introdução descreve o que foi realizado, a Seção 2 descreve os detalhes de implementação, detalhes da representação, *fitness* e operadores utilizados; a Seção 3 é uma análise experimental dos impactos da mudança de parâmetros e a Seção 4 encerra o texto destacando algumas conclusões.

2. Implementação

Esta seção está dividida em duas grandes subseções. A Seção 2.1 apresenta diversas decisões de projeto e inclui instruções para instalação e execução do programa, enquanto a Seção 2.2, discorre em relação às decisões relacionadas à implementação da Programação Genética propriamente dita.

2.1. Arquitetura

A título de completude, esta seção detalha diversos aspectos de *software* da implementação. Primeiramente, a linguagem usada foi Python, versão 3.12.

2.1.1. Instalação

A forma recomendada de instalar as dependências do programa é usando o gerenciador de pacotes uv. Com ele, basta rodar o comando a seguir para instalar as bibliotecas necessárias:

```
uv sync
```

2.1.2. Execução

Existem diversas maneiras de se executar o programa. A principal delas é via linha de comando. No entanto, devido à quantidade de parâmetros do programa, essa opção pode ser massante. De qualquer modo, o comando base, assumindo que o `uv` está sendo usado, é:

```
uv run python -m src ...
```

Em que as reticências denotam os parâmetros. Ao todo, existem 13 parâmetros, majoritariamente obrigatórios. A seguir, estes são listados:

- `--seed`, semente aleatória
- `--leaf-probability`, probabilidade de se gerar uma folha ao construir a árvore
- `--mutation-probability`, probabilidade de mutação
- `--swap-terminal-probability`, probabilidade de se trocar um terminal na mutação
- `--swap-operator-probability`, probabilidade de se trocar um operador na mutação
- `--crossover-probability`, probabilidade de cruzamento
- `--population-size`, tamanho da população

- `--tournament-size`, tamanho do torneio
- `--elitism-size`, tamanho do elitismo (tamanho 0 significa ausência de elitismo)
- `--max-generations`, quantidade de gerações
- `--max-depth`, profundidade máxima da árvore
- Banco de Dados (não possui *flag*). Deve ser especificado como `wine_red` ou `breast_cancer_coimbra`

O parâmetro restante controla se a saída deve ser verbosa ou não. A saída verbosa inclui as "Estatísticas Importantes" da especificação, enquanto que a versão não verbosa apenas imprime a Métrica V dos dados de teste. A corretude dos parâmetros é vastamente checada, por exemplo: o programa retorna uma exceção caso o tamanho do elitismo seja maior do que o número de indivíduos na população.

Devido à quantidade exorbitante de parâmetros, foram desenvolvidas algumas alternativas para execução. Apesar disso, para que elas funcionem corretamente, é necessário ter seguido os passos da Seção 2.1.1. A alternativa mais simples é rodar o programa pelo VS Code, através da configuração de depuração. Ela atribui aos parâmetros valores inspirados na especificação, que descreve parâmetros iniciais para começar os experimentos[1]. Esta foi a principal forma de execução durante o desenvolvimento.

A segunda alternativa recomendada é através do Jupyter. A partir do módulo `loader`, é definida uma configuração padrão para o programa (uma vez que não é possível passar os parâmetros diretamente via linha de comando no Jupyter). A finalidade dessa forma de execução foi permitir a visualização dos operadores genéticos, mas ela também pode ser usada para rodar o programa por completo.

Além disso, o programa também conta com alguns testes de unidade, com uso do *framework* pytest[2]. Enquanto o Jupyter foi usado para testar funcionalidades de "mais alto nível", os testes foram desenvolvidos para garantir que comportamentos mais simples não fossem "quebrados" no decorrer do projeto. Os testes podem ser executados com o comando

```
uv run python -m pytest
```

Para concluir, na fase de experimentação, foram desenvolvidos alguns *scripts* para facilitar a execução repetida do programa. Os *scripts* foram implementados para o *shell* fish e não são compatíveis com o bash e afins. Eles se encontram no diretório `./run`, na raiz do projeto (mais detalhes na Seção 2.1.3).

2.1.3. Estruturação dos Arquivos

O programa está dividido em módulos. A ideia era simplificar o *loop* principal do algoritmo, abstraindo detalhes de implementação. Assim, o *loop* principal chama uma função `select()` (que faz a seleção), implementada em outro arquivo (por exemplo). Essa modularidade torna o programa mais extensível, pois seria relativamente simples trocar as funções que implementam o cerne do algoritmo. Devido à modularização, ao todo são 18 arquivos de código-fonte[3].

---

[1] Por exemplo, probabilidade de cruzamento igual a 90%. Alguns parâmetros foram escolhidos arbitrariamente, por falta de referência. Para detalhes, consultar o arquivo.

[2] Que é uma das dependências do projeto, então não é necessário instalá-lo separadamente.

[3] Excluindo arquivos de teste e afins.

A estruturação do repositório como um todo (para além do código-fonte) também traz essa modularidade. Além do código fonte que se encontra na pasta `./src` (e desta documentação, em `./docs`), o projeto também conta com:

- `./test`, com os arquivos de teste;
- `./scripts`, com alguns programas simples para auxiliar na agregação de estatísticas e geração de gráficos;
- `./run`, com alguns *scripts* para execução do programa (ver final da Seção 2.1.2);
- `./data`, contém os *datasets Breast Cancer* e *Wine Red*;
- `./dumps`, com os relatórios brutos das execuções usadas para a Seção 3;
- `./assets`, contém as figuras e estatísticas agregadas usadas na Seção 3.

Ademais, alguns *notebooks* do Jupyter estão espalhados na raiz.

2.2. Programação Genética

Essa seção descreve as escolhas de implementação associadas à Programação Genética.

2.2.1. Preliminares

Antes de iniciar a discussão sobre a PG, duas decisões cruciais foram tomadas em relação a:

- Normalização dos Dados. Para evitar efeitos provenientes da diferença de escala entre as colunas, os dados foram normalizados. Mais especificamente, foi realizada a norma `Max`, por coluna. Assim, os valores de cada coluna variam entre 0 e 1, sendo que sempre vai existir ao menos um valor 1 por coluna. Outros métodos estatísticos foram considerados, como a transformação de Box-Cox, mas eventualmente foram descartados (por exemplo, o Box-Cox introduz valores negativos que poderiam afetar o desempenho da PG).
- Cálculo das Distâncias. Para simplificar o cálculo da distância, a função gerada pela PG é, primeiro executada para uma instância do *dataset* e depois para outra. A distância final entre as instâncias é o valor absoluto da diferença. A base para essa decisão foi exclusivamente a simplicidade de implementação.

2.2.2. Representação de Indivíduos

Optou-se por representar um indivíduo como uma árvore binária, com suporte às quatro operações básicas (adição, subtração, divisão[4] e multiplicação) e com os terminais sendo as *features* do *dataset*. Tal como a escolha do cálculo das distâncias, essa opção foi guiada pela simplicidade de implementação (apesar de pecar em poder de representação). Somente com os operadores básicos (operadores binários, no geral), os operadores genéticos são vastamente simplificados (mais detalhes na Seção 2.2.5).

A implementação da árvore não dependeu de bibliotecas. A árvore é gerada aleatoriamente (ver Seção 2.2.3) e, para se realizar a avaliação de uma dada coluna do *dataset*, é realizado um caminhamento na árvore, substituindo os terminais pelos valores da coluna. A árvore também conta com algumas operações de auxílio, como as funções `height`, `depth` e `traverse`. Além disso, possui um conversor para representação gráfica usando o Graphviz[5].

[4] Naturalmente que foi usada a divisão protegida, substituindo 0 por um ε.

[5] Na seção de instalação não foi comentado, mas para usar essa representação, é necessário instalar o Graphviz separadamente.

2.2.3. Geração de Indivíduos

A geração de indivíduos também foi a mais simples possível. A abordagem utilizada é similar ao método *grow*, visto em sala de aula. Cada nó tem uma chance de ser um terminal (ou é garantido que será um terminal se for a profundidade máxima), caso contrário é escolhido um operador e seus filhos são gerados recursivamente. Uma abordagem mais balanceada, como o *ramped half-and-half* não foi considerada, por introduzir uma certa complexidade. Similarmente, o método *grow* introduz complexidade pois adiciona mais um parâmetro a mais no algoritmo: a chance de uma folha ser gerada (`--leaf-probability`, da Seção 2.1.2).

2.2.4. Seleção

Como definido na especificação, o método de seleção foi o torneio. A única questão notável aqui foi o esquema de substituição: os pais sempre substituem os filhos. Ou seja, não foi usado *crowding* ou mecanismo similar para lidar com diversidade.

2.2.5. Operadores Genéticos

Foram implementados implementados os dois operadores genéticos tradicionais: cruzamento e mutação (possuindo 4 variantes). Como existem probabilidades associadas a essas operações, é possível que nenhuma delas aconteça e os indivíduos selecionados apenas sejam inseridos novamente na população (também é possível que ambas aconteçam).

2.2.5.1. Cruzamento

O cruzamento implementado seleciona um nó de cada árvore e faz a troca. Como a árvore é bem simples, não é necessário preocupação com a propriedade de fechamento. O único invariante que deve ser respeitado é a profundidade das árvores, que deve ser menor que o tamanho máximo do indivíduo. Para checar isso, as funções auxiliares mencionadas na Seção 2.2.2 são usadas. Mais especificamente, confere-se se a soma da altura de um nó com a profundidade do outro excede o limite. Como a árvore binária possui um nó para indicar o pai, certo cuidado é tomado para realizar corretamente a troca.

As figuras a seguir[6], geradas com base na representação do Graphviz (ver Seção 2.2.2), ilustram um exemplo de cruzamento. Nelas, os nós destacados de vermelho indicam a troca.

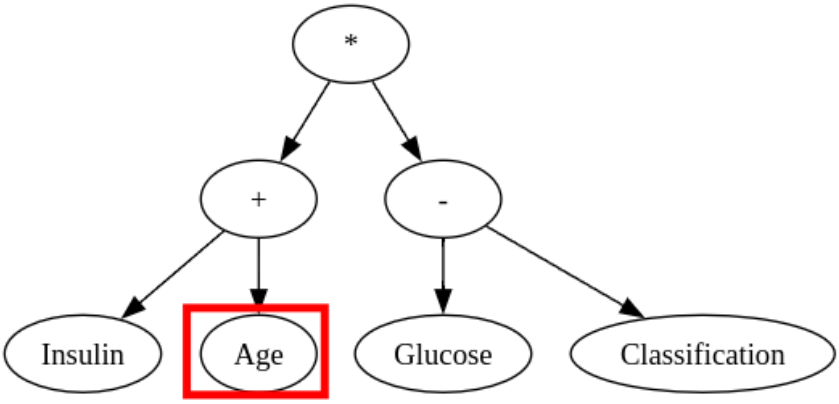

Figura 1: Cruzamento – Pai 1

[6]O exemplo foi didático (isto é, só troca folhas) para que as imagens não ocupassem muito espaço.

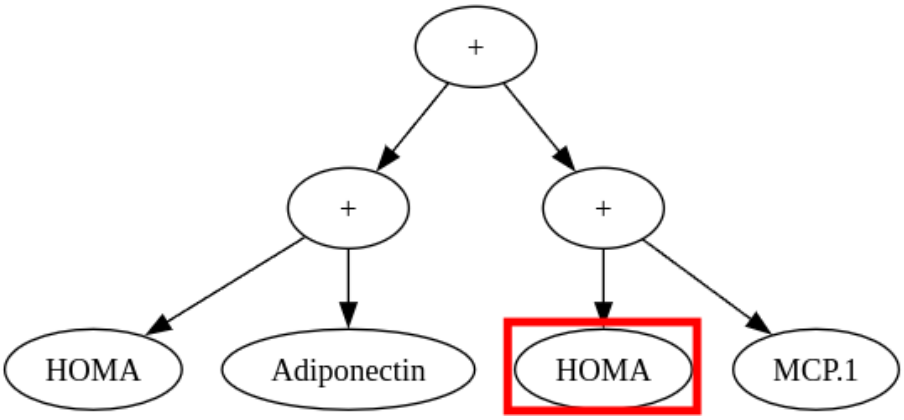

Figura 2: Cruzamento – Pai 2

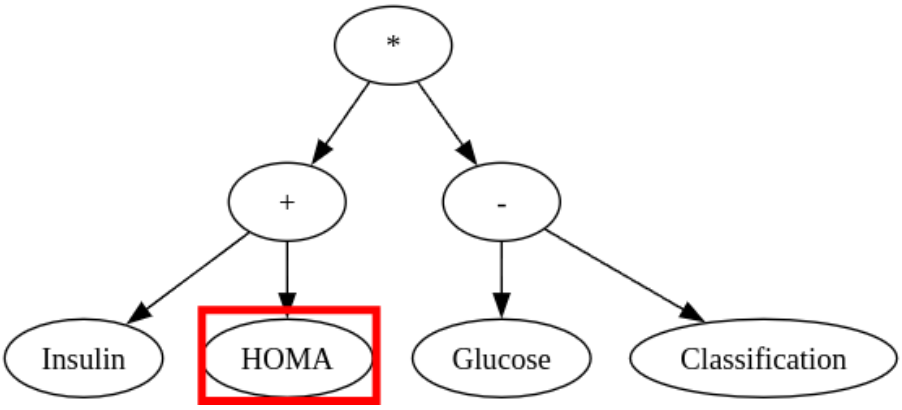

Figura 3: Cruzamento – Filho 1

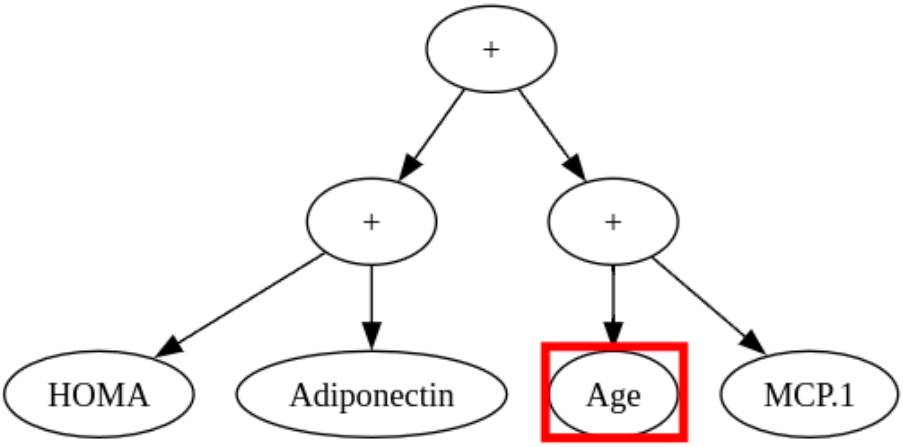

Figura 4: Cruzamento – Filho 2

2.2.5.2. Mutação

A mutação consiste na seleção e troca de apenas um nó da árvore gerada. Como comentado no início da seção, existem 4 variantes: duas ocorrem apenas para nós terminais, enquanto as outras duas apenas para operadores. Ambos os tipos de nós podem ser trocados por outro nó do tipo correspondente. Ou seja, a primeira variante é a troca de um operador por outro e a segunda variante é uma troca de um terminal por outro. É garantido que haverá troca, pois o nó correspondente é excluído da *pool* de seleção.

As outras variantes são: um terminal pode ser expandido em uma nova árvore e um operador pode ser reduzido para um único terminal. Para escolher qual mutação será aplicada a um dado nó, primeiro checa-se se o nó é um operador ou terminal e, em seguida, os parâmetros `--swap-terminal-probability` e `--swap-operator-probability` escolhem entre a troca "comum" ou a variante "especial". Como no caso do cruzamento, o único invariante que pode causar problemas é a altura máxima do indivíduo, que é checada quando a expansão é escolhida: a altura da nova nova árvore é, no máximo, a altura máxima menos a profundidade do nó selecionado. A seguir, as figuras ilustram os diferentes tipos de mutação:

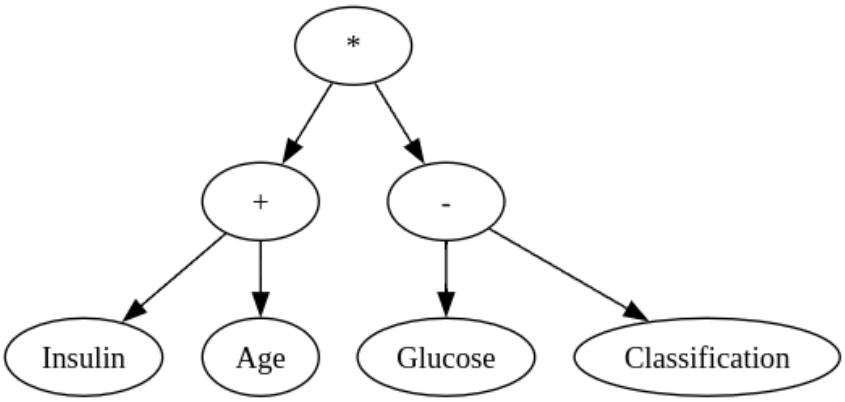

Figura 5: Mutação – Árvore Inicial

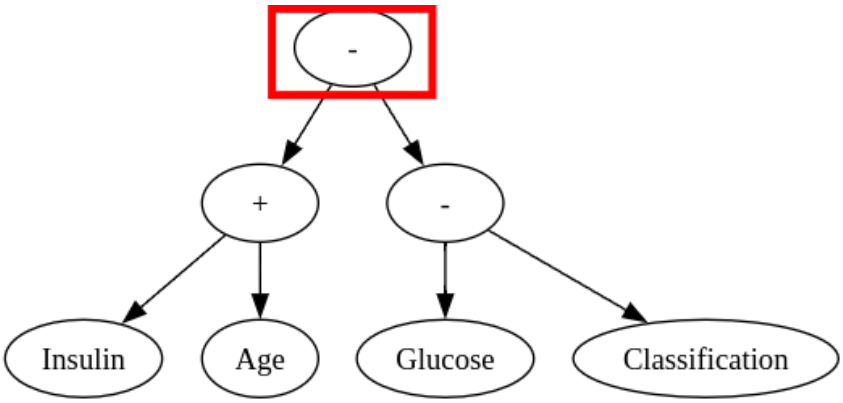

Figura 6: Mutação – Troca de Operador

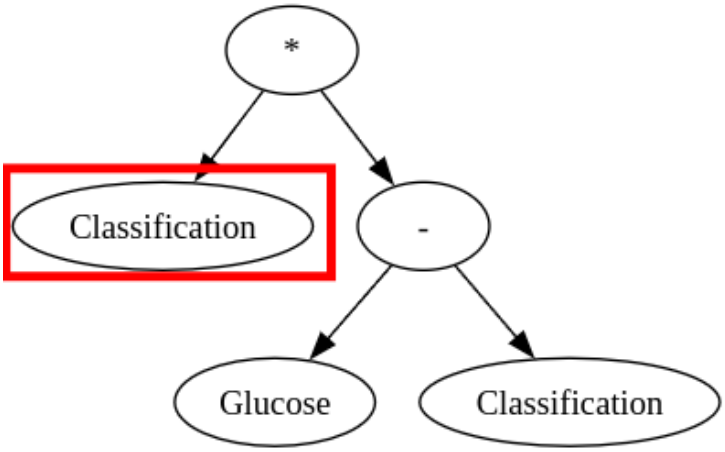

Figura 7: Mutação – Redução de Operador

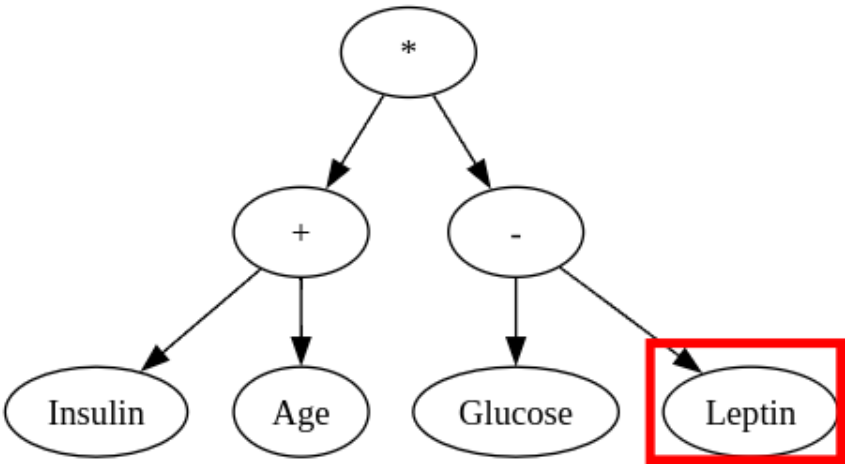

Figura 8: Mutação – Troca de Terminal

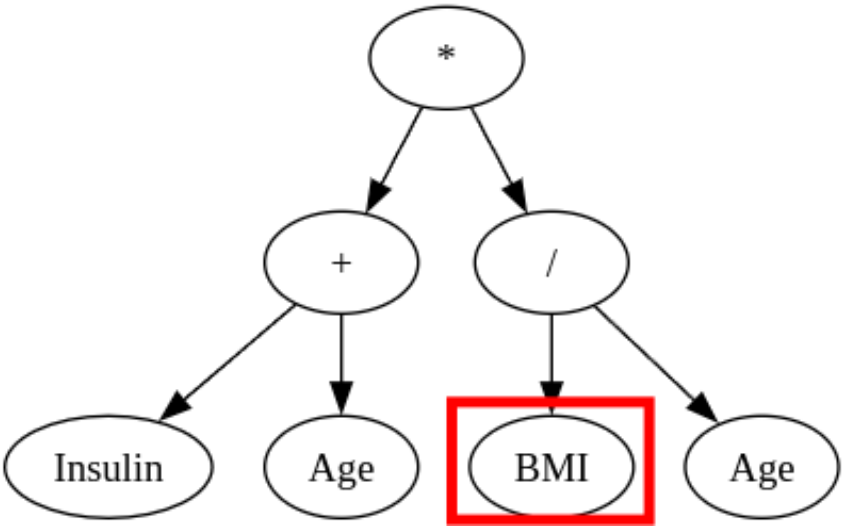

Figura 9: Mutação – Expansão de Subárvore

Como o tamanho máximo do indivíduo foi limitado a 3 na geração destas figuras, a Figura 8 e a Figura 9 demonstram uma grande similaridade, uma vez que a expansão da subárvore foi limitada a apenas um nó. Apesar do ganho em explorabilidade, a diversidade da mutação possui o custo de introduzir dois novos parâmetros.

2.2.6. Fitness

Além do comentário sobre o cálculo das distâncias da Seção 2.2.1, não há muito o que comentar sobre sua implementação. Tentou-se realizar o cálculo das distâncias com programação paralela (via Numba), mas não foram obtidos bons resultados[7]. De qualquer modo, o Numba foi usado para compilar a função de cálculo de distâncias, o que tornou a função mais rápida (a diferença é mais notável para instâncias maiores).

O cálculo da *fitness* funciona da seguinte maneira: primeiro, as avaliações das colunas são geradas, depois é calculada a matriz de distâncias (com base nas avaliações) e é feito um `fit_predict` no `AgglomerativeClustering` do `scikit-learn`. Por fim, é realizado o cálculo da Métrica V, comparando-se com as classes reais.

[7] Outra possibilidade de melhoramento de performance que não se mostrou promissora, foi o uso do compilador pypy.

3. Experimentos

O roteiro de experimentação foi baseado no guia geral da especificação e é sumarizado aqui. Todos os dados gerados na execução dos experimentos se encontram no diretório `./dumps`, como apontado na Seção 2.1.3. Além disso, os experimentos são reprodutíveis, uma vez que a semente aleatória é um parâmetro não somente do próprio programa, como também dos *scripts* de execução (em `./run`). Ainda, quando a *flag* de verbosidade está habilitada, é possível recuperar todos os parâmetros de uma dada execução, pois esse artefato é registrado (e também está presente nos *dumps*).

Cada combinação de parâmetros foi repetida 10 vezes e todos os experimentos foram realizados a partir da modificação de apenas um parâmetro por vez. São analisados, principalmente, os gráficos da evolução da *fitness* (com melhor, média e pior), da quantidade de repetições e do desempenho dos filhos (quantidade melhorada ou não). A escolha dos parâmetros foi feita de forma iterativa: o primeiro parâmetro foi escolhido e fixado, depois o segundo e assim por diante. Uma análise mais detalhada, mas incompatível com o tempo proposto, seria retroativamente tunar os parâmetros: por exemplo, ao definir o terceiro parâmetro, checar se os anteriores ainda são os melhores.

Os experimentos foram realizados na base de câncer de mama (a outra base é testada na Seção 3.2). Como referência para este conjunto de dados, usando-se a distância euclidiana, a Métrica V tem valor igual a, aproximadamente, 0.1256. Em termos de notação, nesta seção $|N|$ é usado para se referir ao tamanho da população, $|G|$ ao número de gerações e $|T|$ ao tamanho do torneio.

3.1. *Tuning* de parâmetros

Esta subseção está dividida da seguinte maneira: a Seção 3.1.1 relata a escolha dos tamanho da população e o número de gerações; a Seção 3.1.2 a escolha de probabilidades de cruzamento e mutação; a Seção 3.1.3 o tamanho ideal do torneio e a Seção 3.1.4 o efeito da presença ou ausência de elitismo. A Seção 3.1.5 discute alguns parâmetros adicionais.

3.1.1. Tamanho da População e Número de Gerações

Primeiramente, foi decidido o tamanho da população, considerando as opções propostas no guia: **30**, **50**, **100** e **500**. Na rodada inicial, o restante dos parâmetros também foi baseado no guia, exceto os parâmetros adicionais do programa, que foram escolhidos arbitrariamente. Para um aprofundamento maior em quais foram os parâmetros inciais, recomenda-se verificar os registros nos *dumps*. A única diferença notável foi o número de gerações, que foi configurado como **100**. A partir dos registros dessa primeira etapa, foram construídos os gráficos a seguir. A média das repetições é destacada, enquanto a região mais clara é o intervalo de 95% de confiança.

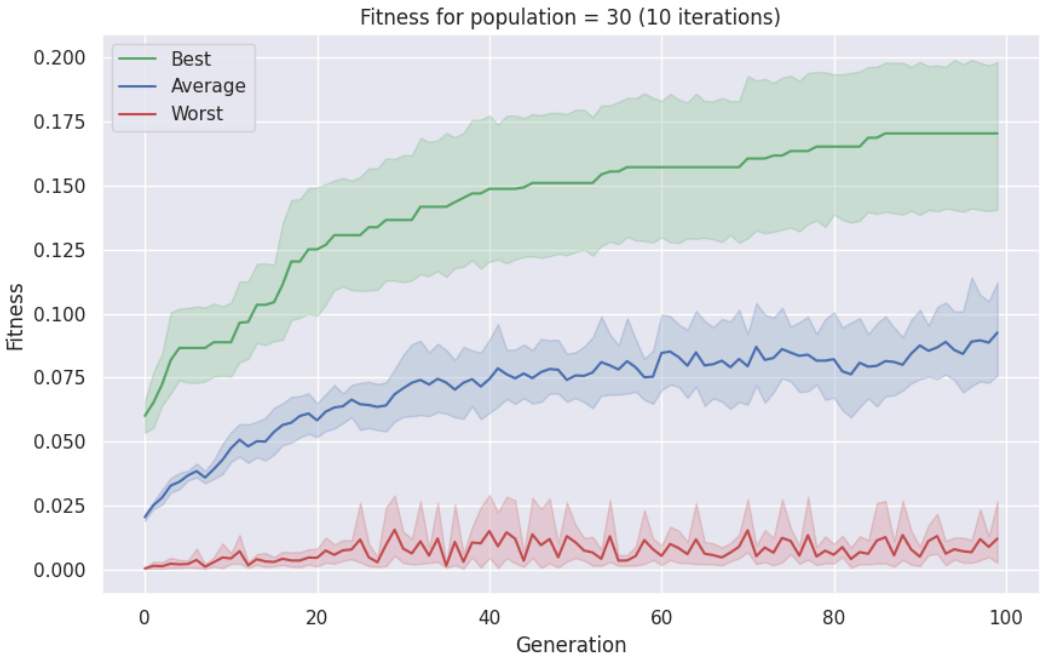

Figura 10: Evolução da *fitness*, $|N| = 30$

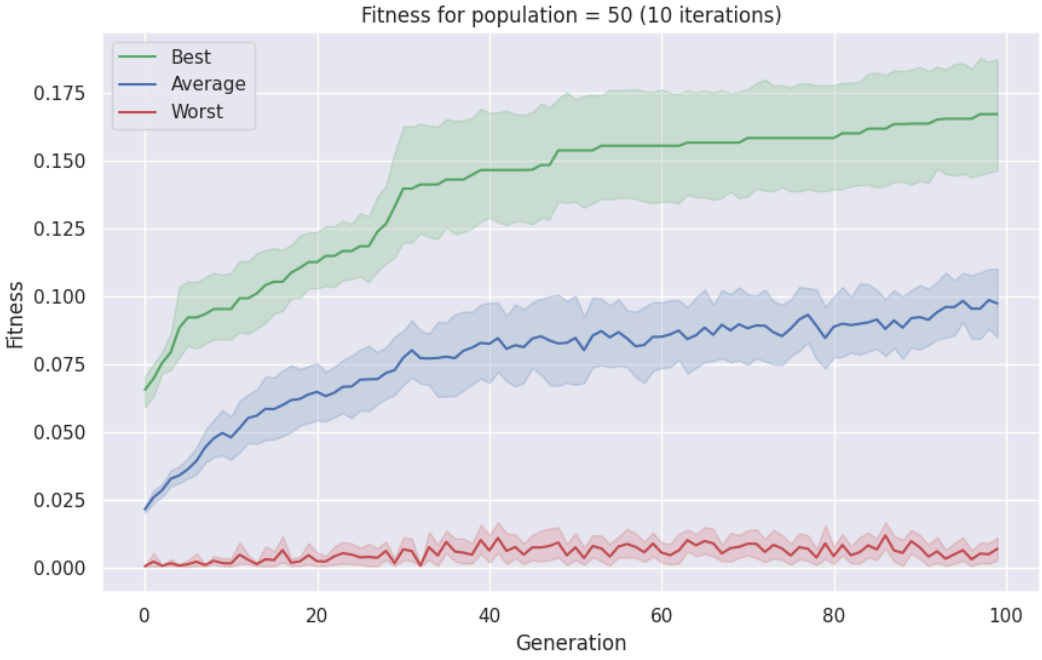

Figura 11: Evolução da *fitness*, $|N| = 50$

Figura 12: Evolução da *fitness*, $|N| = 100$

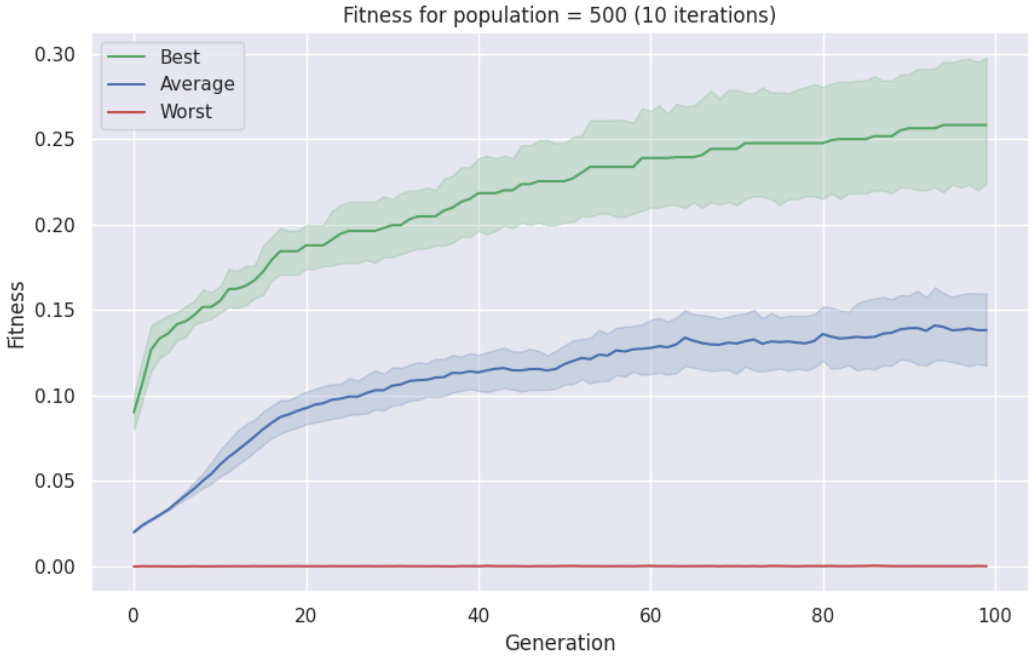

Figura 13: Evolução da *fitness*, $|N| = 500$

Como é possível ver logo na Figura 10, a *fitness* para 30 indivíduos por geração já é superior à *fitness* da distância euclidiana. Também é possível perceber que, felizmente, o formato das curvas está dentro do esperado: nas primeiras gerações, a *fitness* cresce consideravelmente e depois seu crescimento é mais lento. A *fitness* da média da população também tem um bom comportamento: ela não é muito próxima da curva de melhor *fitness*, o que indica que a população não convergiu.

Os gráficos de número de indivíduos repetidos também confirmam isso. Por uma questão de logística, este relatório não inclui todos os gráficos gerados (apesar de estarem disponíveis em `./assets`). De qualquer maneira, a título de completude, segue o gráfico para $|N| = 100$:

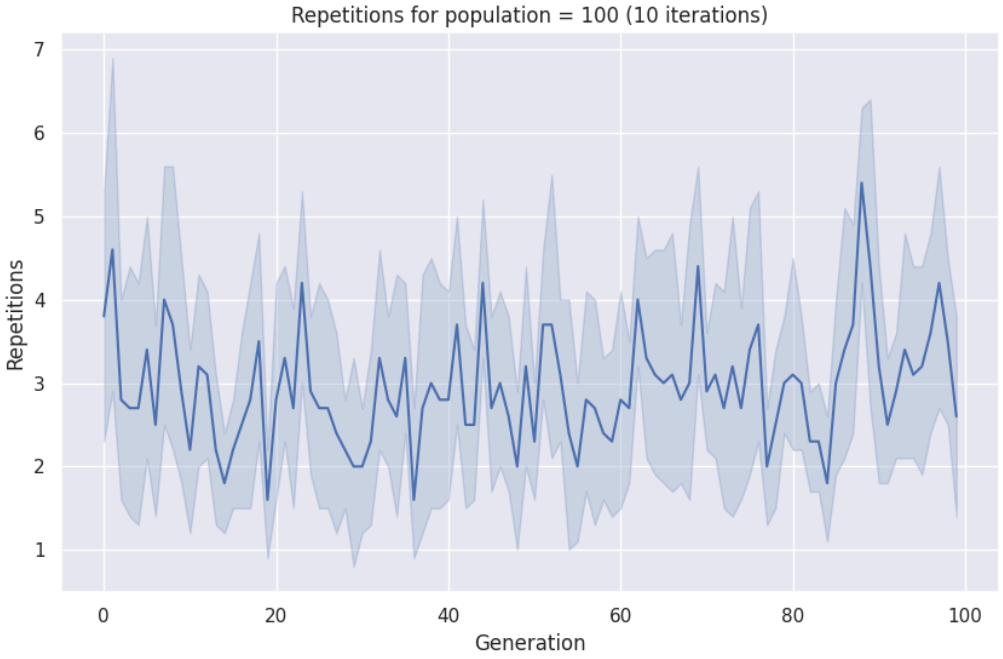

Figura 14: Quantidade de repetições, $|N| = 100$

Parece haver uma grande variância, mas isso é consequência da baixa quantidade de indivíduos repetidos. Como a pressão seletiva é baixíssima na configuração padrão ($|T| = 2$), o resultado é condizente. Tendo em vista, pela Figura 13, que $|N| = 500$ apresentou melhor resultado de *fitness* ($\simeq 0.25$), a um tempo de execução não muito caro ($\simeq 4$ minutos por cada repetição), a escolha para o tamanho da população inicialmente seria 500. No entanto, apesar do tempo relativamente curto para uma única execução, considerando-se o tempo de experimentação como um todo, **foi preferido o tamanho de população igual a 100**.

Uma situação semelhante aconteceu com a quantidade de gerações. Foram consideradas as quatro opções do guia: **30**, **50**, **100** e **500**. Com o número de indivíduos fixo em 100, foram realizadas 10 repetições para 500 gerações. A partir dessa rodada, é possível induzir as métricas para as outras quantidades de gerações.

Pela Figura 15, a quantidade de gerações possui um grande impacto na *fitness*. A primeira linha amarela é o corte para 30 gerações; a segunda o corte para 50 e, a última, o corte para 100. Existe um salto relativamente grande entre 30 gerações e 100 gerações e, considerando o custo computacional, faz sentido preferir a opção maior. Esse não é o caso com a variação de 100 gerações para 500, que é muito cara (especialmente considerando o contexto de múltiplas execuções dos experimentos), apesar da boa melhora em desempenho (de $\simeq 0.214$ a $\simeq 0.2482$). Assim, **foi selecionada a quantidade de gerações igual a 100**.

Devido ao *tuning* dos outros parâmetros não ter sido realizado a essa altura, os gráficos de relação com os filhos e quantidade de repetições ainda não trazem *insights* e, por isso, foram omitidos (apesar de estarem disponíveis no repositório).

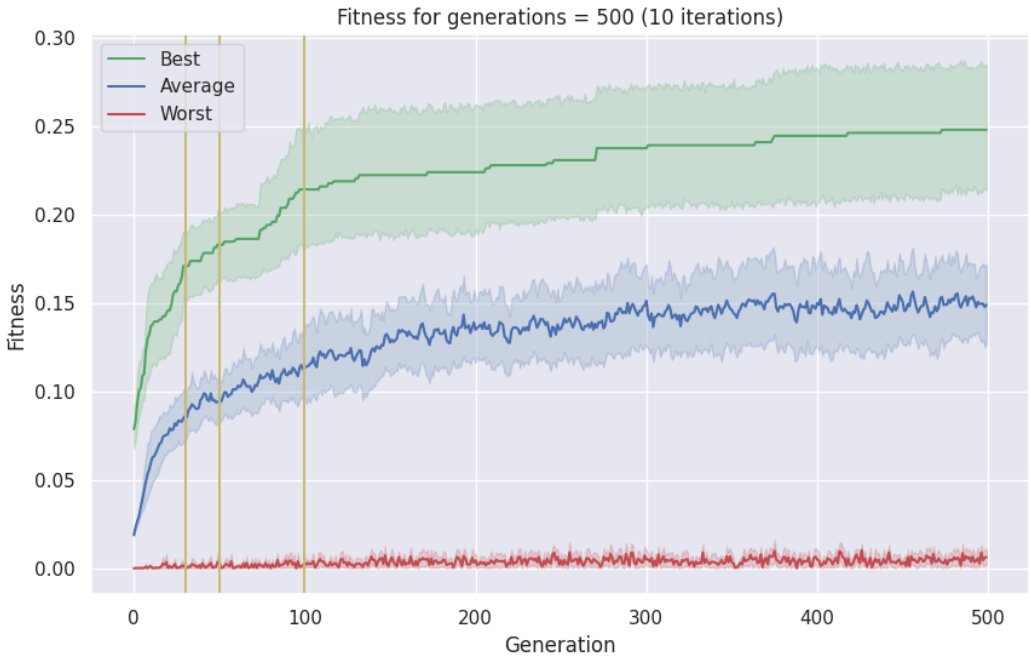

Figura 15: Evolução da *fitness*, até $|G| = 500$

3.1.2. Probabilidades de Cruzamento e Mutação

Com o tamanho da população fixo em 100 e o número de gerações também fixo em 100, foram realizados alguns experimentos para extrair as melhores probabilidades de cruzamento e mutação. As probabilidades relacionadas às variantes de mutação são discutidas na Seção 3.1.5. A configuração "padrão" (usada nos experimentos até então) possui alta taxa de cruzamento (0.9) e baixa taxa de mutação (0.05). Foram realizados testes para 3 outras combinações:

- Baixa taxa de cruzamento (0.6), mantendo a taxa de mutação;
- Alta taxa de mutação (0.3), mantendo a taxa de cruzamento;
- Ambas as mudanças anteriores, simultaneamente.

A evolução da *fitness* da configuração padrão pode ser vista na Figura 12, com *fitness* média final aproximadamente igual a 0.207. Ver os comentários da seção Seção 3.1.1 sobre as outras estatísticas. A seguir, as combinações são avaliadas uma a uma, em ordem, a partir dos gráficos gerados.

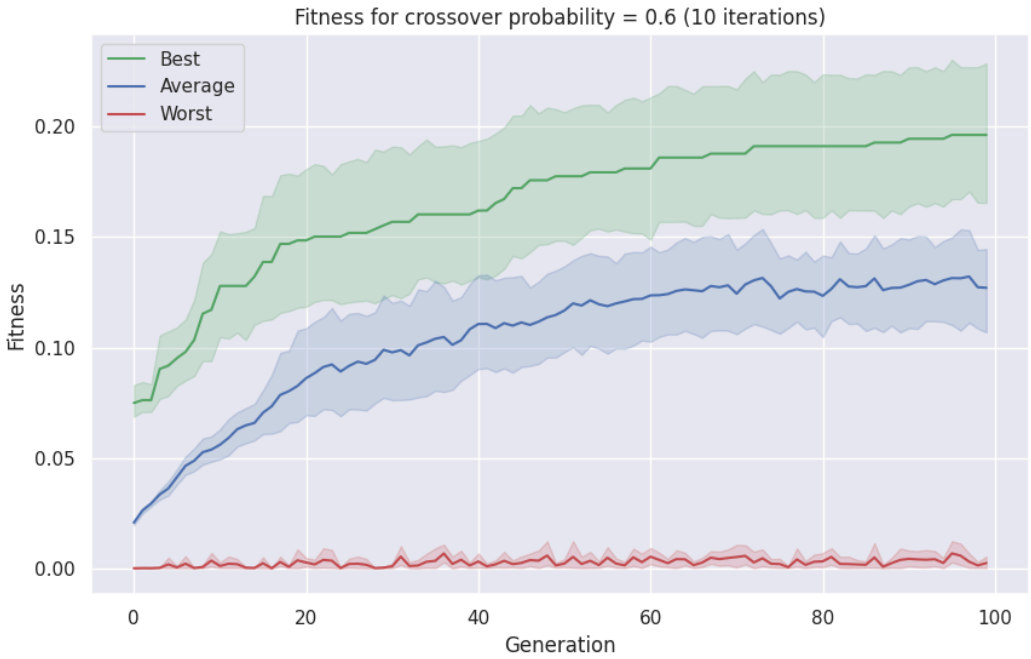

Figura 16: Evolução da *fitness*, *crossover* baixo

É perceptível que o desvio padrão aumentou em relação à configuração padrão. Além disso, o desempenho médio foi um pouco abaixo da configuração padrão, não atingindo 0.2. Analisando a performance dos filhos, é possível perceber nitidamente que a diversidade caiu (em alguns casos, metade da população está repetida) e a há mais dificuldade da busca ser "guiada" para uma direção melhor.

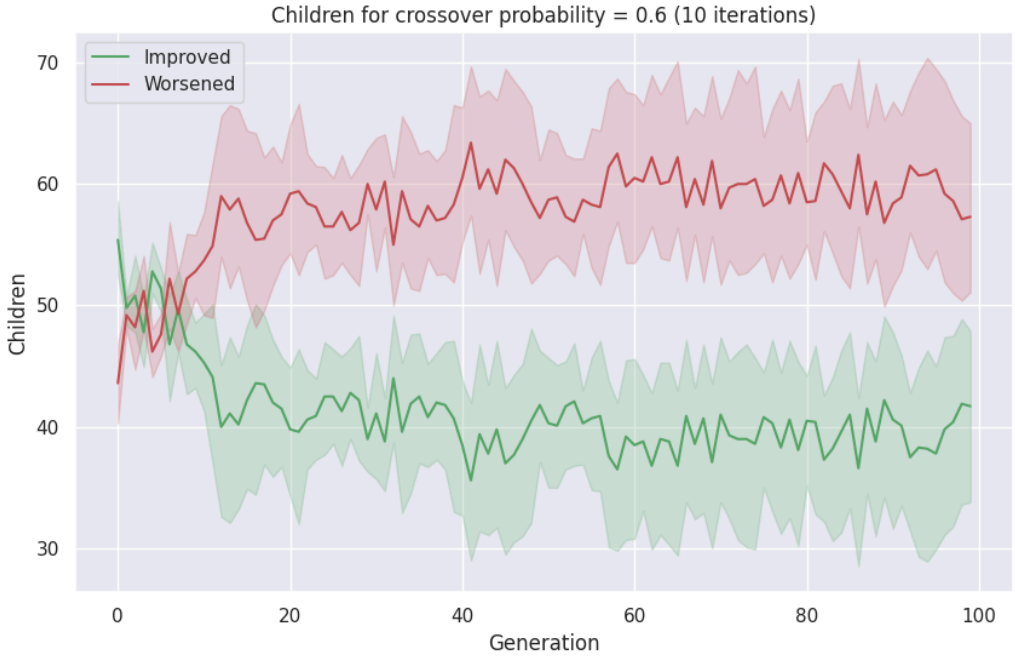

Figura 17: Evolução dos filhos, *crossover* baixo

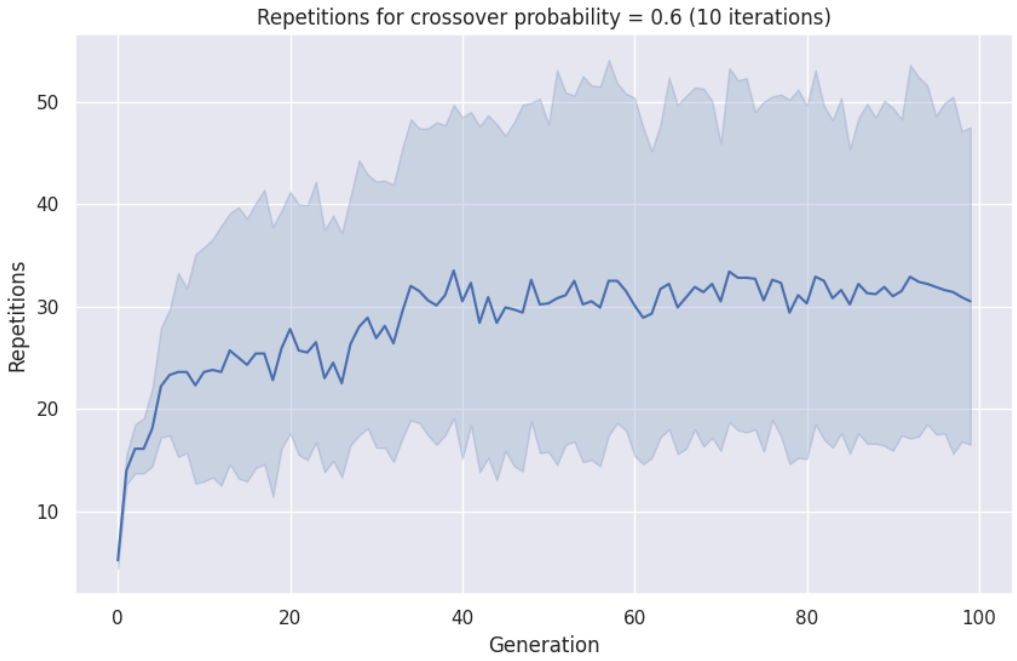

Figura 18: Quantidade de repetições, *crossover* baixo

Variando apenas a mutação, a história é diferente. A *fitness* original foi batida (embora não por muito): o valor ficou como 0.216, na média. No entanto, o *plot* da evolução dos filhos continuou completamente caótico e, devido à grande chance de alteração, o número de indivíduos repetidos foi extremamente baixo, beirando apenas uma ou duas repetições.

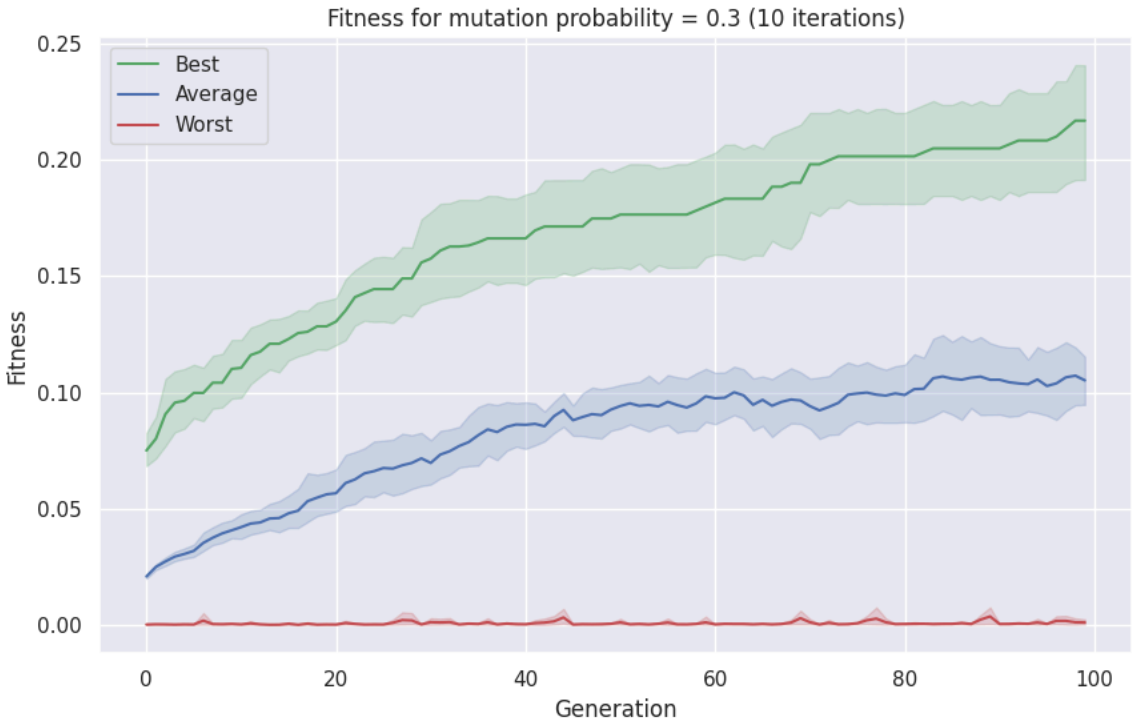

Figura 19: Evolução da *fitness*, mutação alta

Por fim, ambas as modificações foram aplicadas, para realizar a comparação "cruzada". Em resumo, não houve melhora, com média 0.20. De fato, os efeitos observados foram uma combinação dos casos anteriores: o comportamento dos filhos foi consistente em piorar e houve baixa repetição, por volta de 8 indivíduos (as figuras estão disponíveis no repositório). Então, a decisão foi **manter a taxa alta de cruzamento (0.9) mas aumentar a taxa de mutação para 0.3**.

3.1.3. Tamanho do Torneio

3.1.4. Elitismo

3.1.5. Miscelânea

3.2. Teste

4. Conclusões

5. Bibliografia