Trabalho Prático 1

Felipe Cadar Chamone

September 2020

1 Introdução

O principal objetivo deste trabalho é desenvolver conceitos chave para a construção de soluções para problemas usando Programação Genética (PG). Aqui tentaremos resolver um problema de regressão simbólica, que se trata de encontrar uma expressão simbólica que aproxime das amostras fornecidas de uma função desconhecida.

Mais especificamente, tentaremos aproximar 5 funções diferentes. Como o algoritmo possui uma boa variedade de parâmetros, foi feita uma análise de sensibilidade de parâmetros para otimizar o resultado em cada conjunto de dados.

2 Implementação

Aqui descrevemos alguns detalhes de implementação como a representação dos indivíduos, fitness e operadores usados.

2.1 Representação

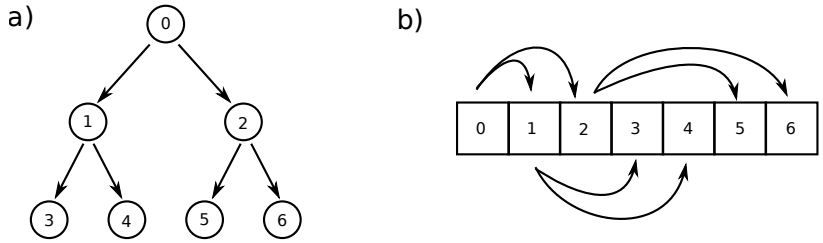


Figure 1: a) Representação de uma árvore binária. b) Estrutura de dados utilizada para implementar a árvore binária.

A especificação do trabalho pede que representemos os indivíduos como árvores, de modo que os nós intermediários são operações e os nós folhas podem ser constantes ou incógnitas.

Para a implementação dessa árvore temos várias opções de estruturas de dados. Por questão de desempenho eu decidi usar uma vetor linear onde para cada nó de posição n, seus filhos vão estar na posição $(2 * n) + 1$ e $(2 * n) + 2$. Dessa forma posso alocar a memória máxima necessária para cada indivíduo na sua criação, deixando mais rápidas as operações genéticas.

Para representar as operações, constantes e incógnitas dentro da árvore, foram usados número inteiros que podem ser mapeados para cada elemento no momento de calcular a fitness. Por exemplo: se temos o mapeamento $1 \leftarrow soma$, $2 \leftarrow X_1$, $3 \leftarrow X_2$, e o vetor $[1, 2, 3]$, geramos a expressão $X_1 + X_2$. Vale a pena lembrar que como agora o genótipo pode ser representado por números inteiros, posso economizar memória alocando vetores to tipo *int* de 8 bits, o que me da 256 valores para mapear entre operações, constantes e incógnitas, amenizando o fato de que essa estrutura de dados vai sempre ocupar uma quantidade máxima de memória mesmo se apenas um nó for utilizado.

Apesar de ser uma estrutura muito rápida, de simples acesso e previsível, ela tem a limitação de que todo nó deve ter apenas 2 filhos. Isso significa que estamos limitados a usar operadores de no máximo 2 entradas, o que não vai representar um grande problema como viremos na próxima sessão.

2.2 Operações

Depois de escolher a estrutura responsável por armazenar os indivíduos, podemos pensar em como implementar os operadores genéticos. Neste trabalho foram usados 3 operadores: Cruzamento, Mutação de ponto e Mutação de subárvore.

2.2.1 Cruzamento

Por se tratar de árvores, o cruzamento foi implementado de forma a trocar subárvores entre dois pais, gerando dois novos indivíduos. Primeiro escolhemos um nó aleatório, com exceção da raiz, do primeiro pai, em seguida escolhemos do segundo pai um nó aleatório tal que nenhum dos dois filhos excedam a profundidade máxima da população.

2.2.2 Mutação

Com o intuito de aplicar *exploitation* e *exploration* em diferente momentos, foram implementados dois tipos de mutação.

A mutação de ponto é responsável pela parte de *exploitation* por alterar apenas um nó da árvore por outro equivalente. Trocamos funções por funções, variáveis por variáveis e constantes por contantes.

A mutação de subárvore é responsável pela parte de *exploration* por escolher um nó aleatório da árvore, com exceção da raiz, e gerar uma nova subárvore aleatória no lugar.

2.3 *Exploitation* vs. *Exploration*

Como estudado em sala, temos essencialmente dois tipos de busca. *Exploration* que explora mais o espaço de soluções para gerar indivíduos mais distantes, fugindo de mínimos locais. *Exploitation*, por outro lado, tem como objetivo otimizar uma parte específica do espaço de busca para de fato chegar ao mínimo local.

A estratégia adotada foi utilizar um parâmetro de exploração. Esse parâmetro define a probabilidade de escolha entre os dois tipos de mutação. Esse parâmetro é iniciado valendo 1 e vai diminuindo linearmente ao passar das gerações.

2.4 Conjunto de funções

Idealmente, nosso conjunto de funções devem possuir algumas propriedades, são elas: Fechamento, Parcimônia e Suficiência. Com a intenção de cumprir esse objetivos usamos apenas as funções e adição $+, -$ e multiplicação $*, \div$. Com esse conjunto conseguimos garantir a propriedade de fechamento com muita facilidade. As propriedades de Parcimônia e Suficiência são um pouco mais complicadas por dependerem do conjunto de treino, mas podemos argumentar que variando o tamanho da árvore o algoritmo genético será capaz de compor funções mais complexas quando necessário a partir do conjunto $+, -, *, \div$. Um detalhe da função $\div$ é que era representa a divisão protegida, dessa forma não temos problemas com divisão por zero.

Além das funções, também inserimos constantes para ocuparem nós folha. Essa parte é muito importante para a composição de funções mais complexas sendo que todas as funções básicas requisitam duas entradas. Para as constantes, usamos números inteiros de 0 a 9. Como seria um pouco complicado para compor constantes conhecidas como π e e, também adicionamos essas duas.

2.5 *Fitness*

Para a escolha das fitness, tínhamos 3 opções: Soma dos erros absolutos, média do erro quadrado (MSE) e raiz da média do erro quadrado (RMSE). A primeiro momento foram implementados os 3 tipos de Fitness, mas a soma dos erros absolutos se mostrou muito instável, já que poucas amostras erradas podiam descartar bons indivíduos. Por esse motivo foi escolhida a MSE. A fitness RMSE também é uma boa candidata, mas a MSE é mais rápida de se calcular.

2.6 População Inicial

A geração da população inicial é muito importante por ser o ponto de partida do algoritmo. Aqui foram implementados 4 tipos de inicializadores de população e a escolha foi feita por análise de sensibilidade em cada dataset.

Além dos clássicos "*Full*", "*Grow*" e "*Ramped Half and Half*" também foi implementado um 4°tipo de inicialização inspirado no *grow* com a intensão de aumentar a probabilidade de árvores maiores, mas sem torna-las obrigatoriamente completas. Nesse método possuímos a probabilidade de um novo

nó ser uma folha, e essa probabilidade cresce linearmente no intervalo da altura máxima da árvore. Com isso a probabilidade de um nó ser folha é $p = (MaxDepth - NodeLevel)/MaxDepth$

3 Experimentos

Para otimizar nossos resultados, precisamos fazer uma análise de sensibilidade de parâmetros. Aqui serão mostradas as análises de 5 parâmetros para cada base de dados. Todos os gráficos mostrados aqui são médias de 30 execuções para cada mudança de parâmetro.

Os parâmetros padrões para a execução dos experimentos são:

- Gerações: 1000
- População: 50
- População Inicial: *Ramped Half and Half*
- Fitness: MSE
- K: 20
- Tamanho máximo da árvore: 7
- Elitismo: Com elitismo
- Alpha: 0.9

Vale observar que o $K = 20$ não foi intencional, mas sim um *typo*. O objetivo era ter usado $K = 2$, mas os experimentos ocupam um tempo muito grande e não foi possível refaze-los.

3.1 Concrete

Essa base de dados é a mais complexa de todas, ela possui 8 variáveis e 1030 amostras.

3.1.1 Tipo de População Inicial

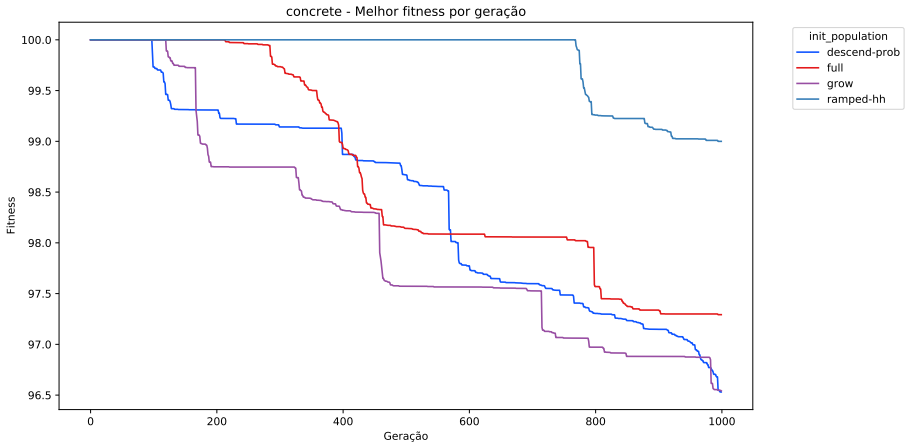


Figure 2: Melhor fitness por geração usando cada um dos 4 métodos de inicialização de população

Na figura 2 podemos notar que o método que mais se distanciou foi o *Ramped*, isso aconteceu provavelmente por forçar indivíduos de pequenos, que não são bons para esse dataset tão complexo. Apesar do empate no final, foi escolhido o método *Grow* por ser o melhor em boa parte do tempo.

3.1.2 Número de gerações

Aproveitando a figura 2, considerando que escolhemos o método *Grow*, podemos observar que a partir da geração 800 não tivemos grandes avanços, por isso escolhemos 800 como quantidade máxima de gerações.

3.1.3 Tamanho da população

Como dito antes, esse dataset é bem complexo, então ele se beneficia muito de uma população maior. Como podemos ver no gráfico da figura 3, a população de tamanho 500 trouxe bons resultados.

3.1.4 Mutação vs Crossover

Esse parâmetro é definido pelo alpha, que é a probabilidade de Crossover. Na figura 4 podemos ver um comportamento que não era o esperado, com o alpha

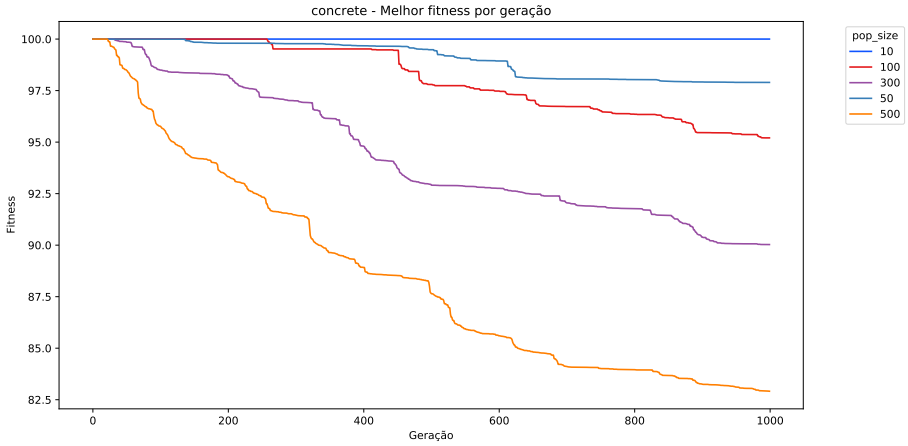


Figure 3: Melhor fitness por geração usando diferentes tamanhos de população

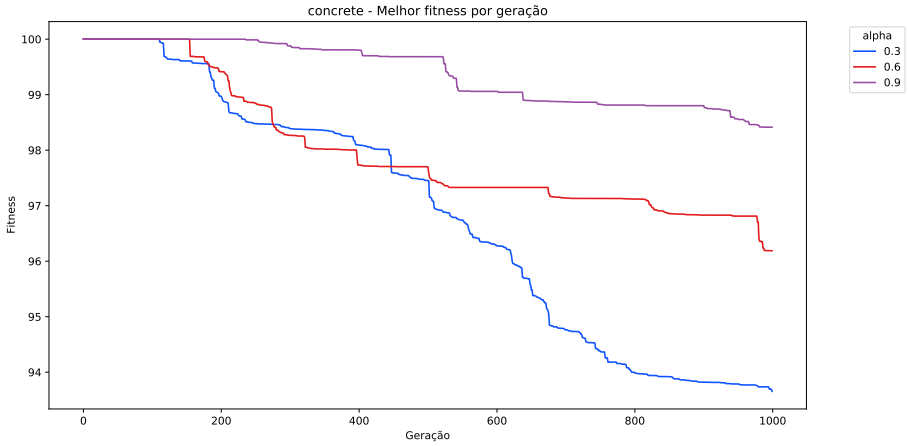


Figure 4: Melhor fitness por geração usando diferentes valores de alpha

= 0.3 sendo o melhor. Considerando que nessa implementação temos dois tipos de mutação e que ao final usamos mais mutações de ponto, faz um sentido que o algoritmo se beneficiou mais de uma busca exploratória no inicío com o alpha pequeno e foi forçado ao *exploitation* ao final independente do alpha.

3.1.5 Torneio

No gráfico da figura 5 podemos observar um resultado interessante. Os valores de K 2 e 10 estão muito próximos, mas o valor 5 representa uma média bem melhor. Podemos argumentar que $K = 2$ tem uma baixa pressão seletiva e falha em selecionar bons indivíduos entre os já existentes e $K = 10$ falha em

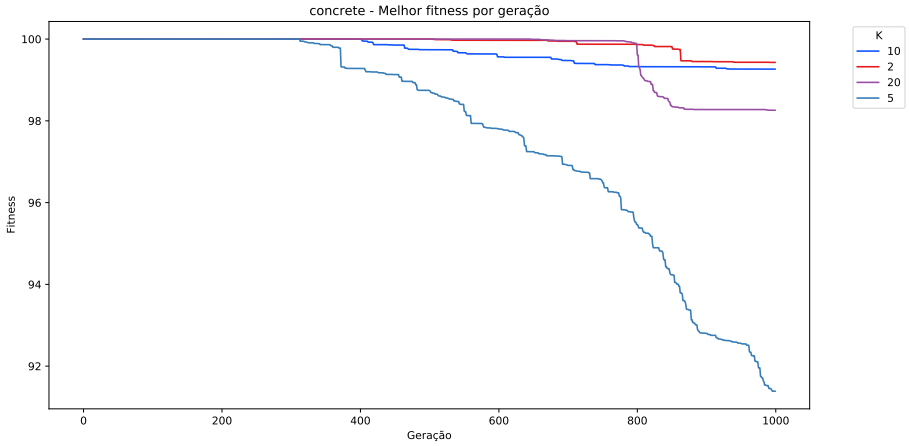


Figure 5: Melhor fitness por geração usando diferentes valores de K do torneio

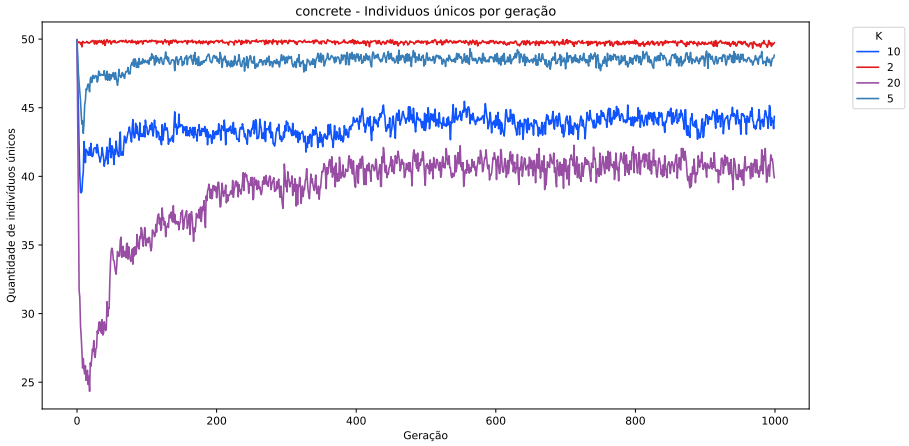


Figure 6: Quantidade de indivíduos únicos usando diferentes valores de K do torneio

permitir uma maior variedade pela alta pressão seletiva. Esse comportamento pode ser observado também no gráfico da figura 6, onde os valores de K para 2 e 5 produzem quantidade próximas de indivíduos únicos e $K = 10$ já reduz suficientemente esse número.

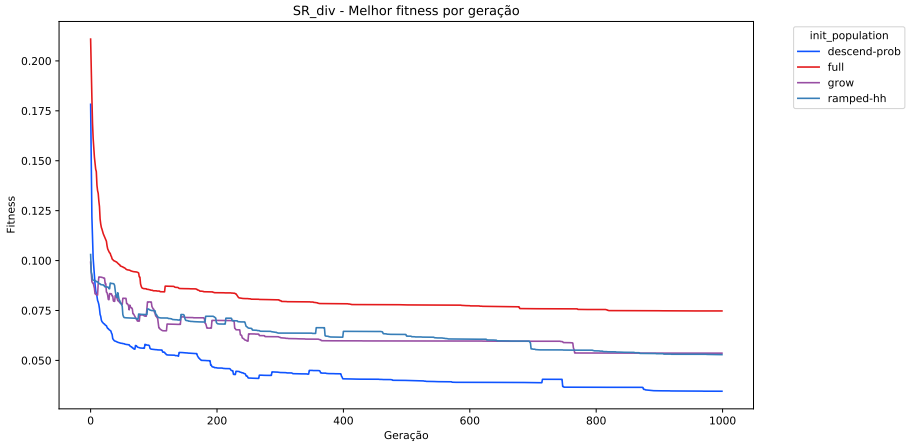


Figure 7: Melhor fitness por geração usando cada um dos 4 métodos de inicialização de população no dataset SR-div

3.2 SR-Div

3.2.1 Tipo de População Inicial

O SR-Div é um dataset bem fácil e simples, então independente da escolha de método de inicialização, a fitness fica bem pequena. Já que temos que fazer uma escolha, vamos com a melhor inicialização do gráfico, o *"Descend-Prob"*.

3.2.2 Número de gerações

Aproveitando o gráfico 7, podemos ver que em todas as linhas praticamente não ha melhora depois de 100 gerações. Por isso vamos mater 100 como numero máximo de gerações.

3.2.3 Tamanho da população

Assim como no gráfico de inicialização de população, todas as linhas do gráfico de tamanho da população, figura 8 estão bem próximas. Esse dataset também costuma convergir bem rápido, então optamos por escolher 500 indivíduos para maximizar a exploração no espaço de busca no começo do algoritmo.

3.2.4 Mutação vs Crossover

Novamente, como já foi explicado na sessão 3.1.4, o algoritmo preferiu um baixo valor de alpha pela existência do parâmetro de exploração.

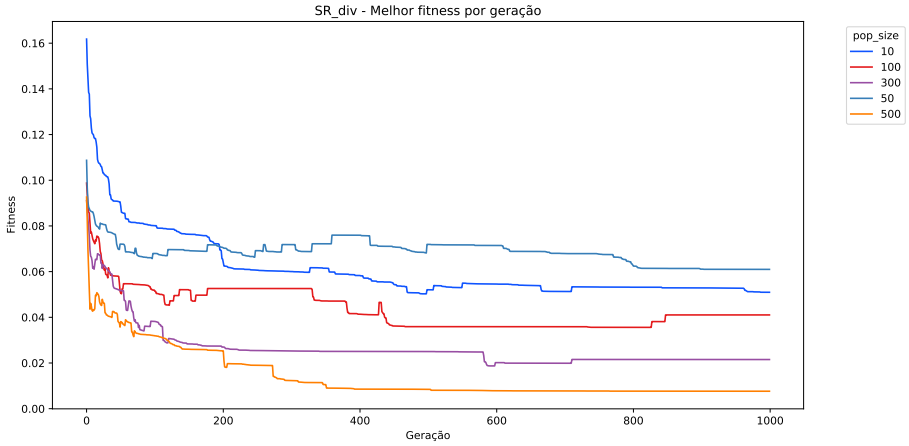


Figure 8: Melhor fitness por geração usando diferentes tamanhos população no dataset SR-div

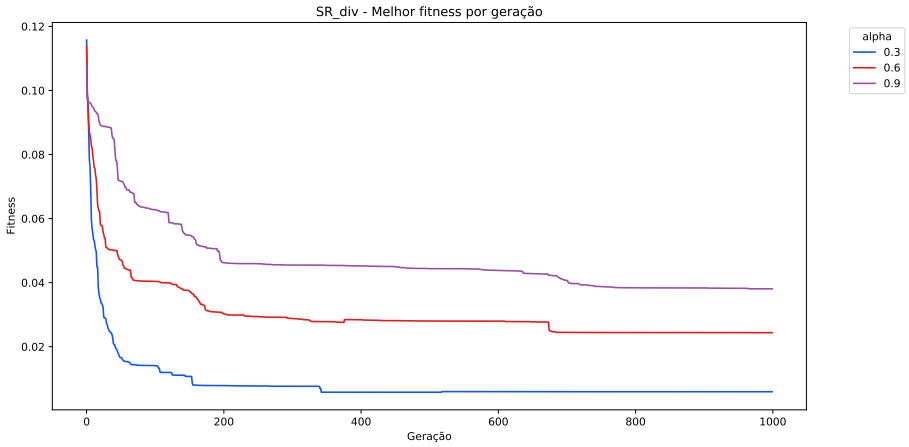


Figure 9: Melhor fitness por geração usando diferentes valores de alpha no dataset SR-div

3.2.5 Torneio

Como mostrado no gráfico da figura 10, o $K = 2$ foi o que mais prolongou a convergência e, ao mesmo tempo, obteve o melhor resultado de melhor fitness.

	Concrete	SR-Div	SR-Div Noise	SR-Circle	SR-Ellipse Noise
Tipo de População Inicial	Grow	Descend Prob	Grow	Grow	Grow
Número de gerações	800	100	100	100	100
Tamanho da população	500	500	500	50	10
Alpha	0.3	0.3	0.6	0.9	0.6
Torneio	5	2	2	2	5

Table 1: Tabela final comparando a análise de sensibilidade de parâmetros de todos os datasets

3.3 Outros datasets

Como especificado na documentação do trabalho, poderia ser escolhido um dataset entre os Toys para representar todos eles, mas pelas diferentes complexidades de cada um deles rodamos os mesmos testes nos outros 3 datasets e, de maneira breve, mostrarei os resultados aqui.

Na tabela 1 podemos ver que os datasets se dividem em dois grupos. O SR-Div e o SR-Div Noise estã relativamente próximos, e as alterações só permitiram um pouco mais de diversidade inicial e mais utilização de crossover. O SR-Circle e SR-Ellipse Noice tabém estão próximos e provavelmente possuem uma complexidade semelhante.

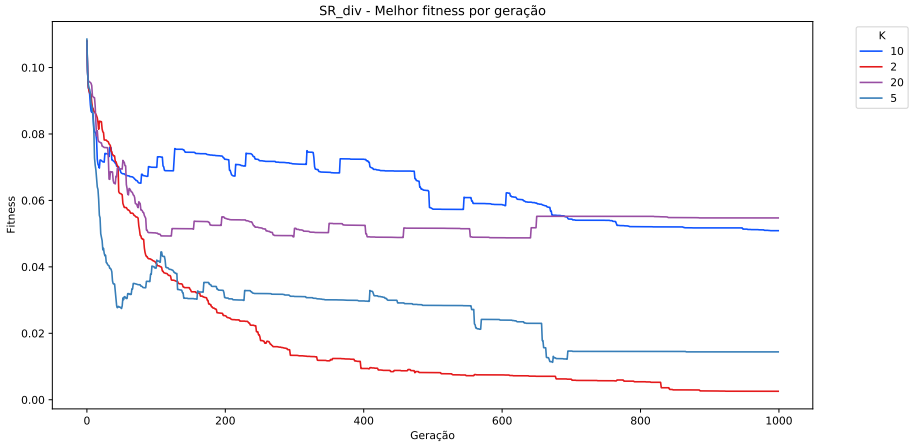


Figure 10: Melhor fitness por geração usando diferentes valores de K do torneio no dataset SR-div

4 Conclusão

Usando os parâmetros obtidos na análise experimental, tabela 1, rodamos cada datasets mais 30 vezes e os resultados estão a seguir.

4.1 Concrete

No gráfico da figura 11 podemos observar que o algoritmo convergiu para uma solução, apesar de não possuir uma fitness muito boa. A partir desse gráfico podemos também inferir que as funções usadas não são suficientes para representar essa base de dados. Outro dado interessante é a porcentagem de filhos de mutação e cruzamento são melhores que seus respectivos pais, como é mostrado no gráfico 13. Nele podemos ver que a partir da geração 400, é mais vantajoso fazer cruzamento do que mutação.

Analisando também a quantidade de indivíduos únicos, podemos observar uma queda, indicando que o algoritmo está convergindo.

4.2 Outras Bases

As outras bases são mais simples e podemos visualizar seus resultados diretamente. A figura 14 mostra o melhor resultado de cada um das outras 4 bases de dados.

Podemos observar que as funções utilizadas não foram suficientes para representar bem os dados do SR-circle e SR-Ellipse-noise, mas que o algoritmo conseguiu reduzir ao máximo as distancias entre os pontos reais e preditos.

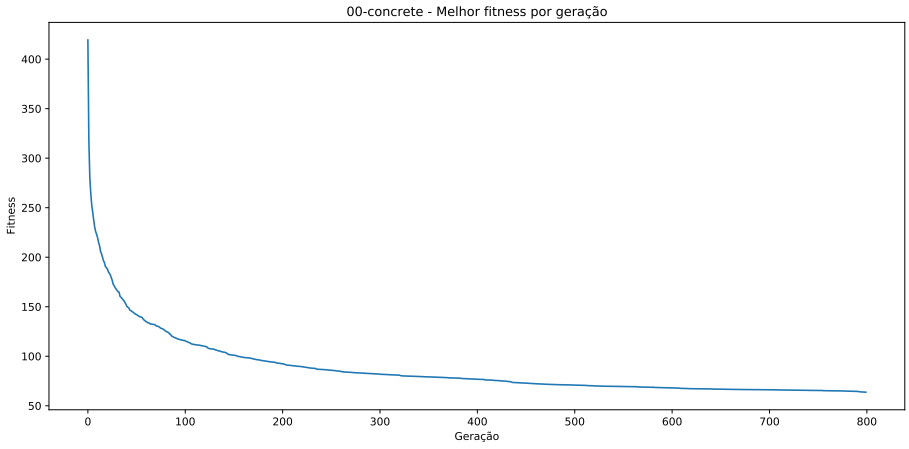


Figure 11: Média das melhores fitness por geração no dataset concrete

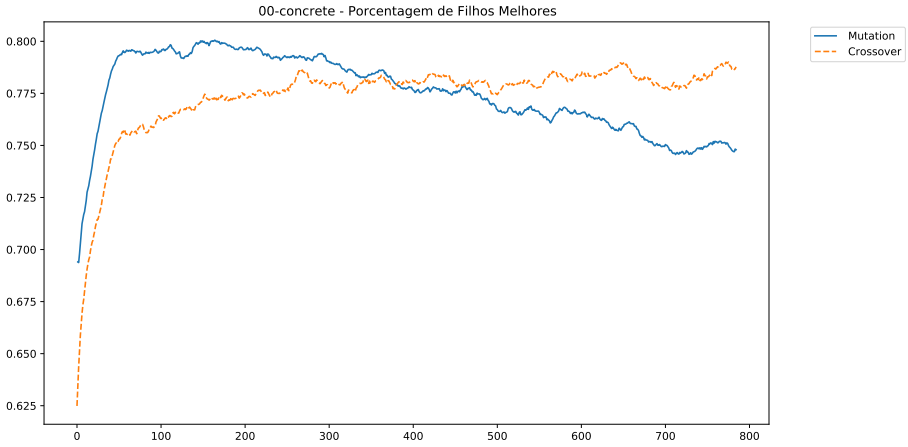


Figure 12: Porcentagem de indivíduos melhores que seus respectivos pais em cada geração.

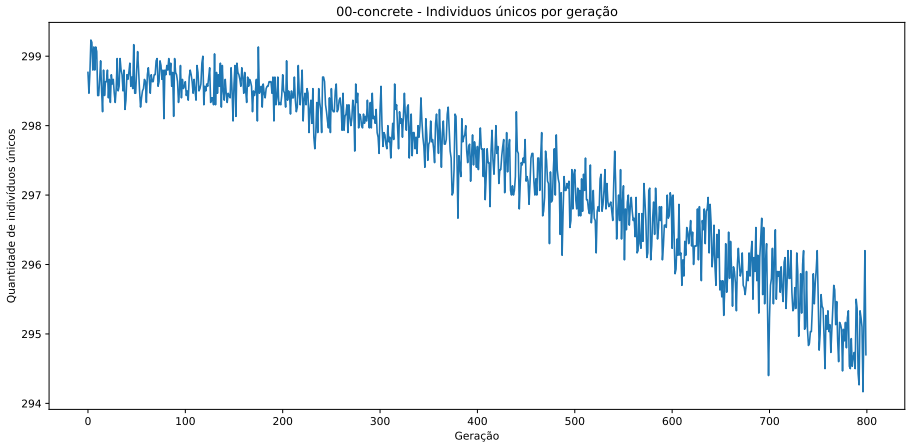
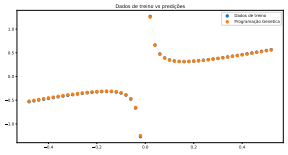
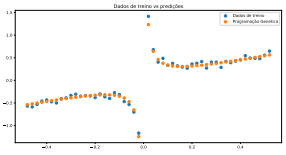


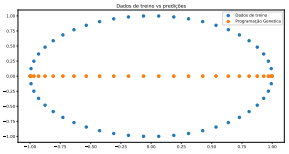
Figure 13: Quantidade de indivíduos únicos por geração.



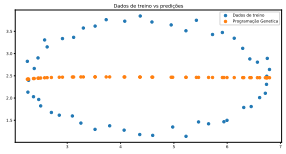
(a) Resultado do algoritmo na base SR-div



(b) Resultado do algoritmo na base SR-div-noise



(c) Resultado do algoritmo na base SR-circle



(d) Resultado do algoritmo na base SR-ellipse-noise

Figure 14: Conjunto de resultados nas bases toy.