Relatório TP1

Diogo Neiss

Introdução

Para a resolução do problema de regressão simbólica estudei os conceitos de *Structred Grammar Evolution* para implementação do trabalho. Para isso, gerei uma gramática base, sem recursão, e criei regras para expandir de acordo com a profundidade desejada.

Inicialmente tentei fazer a versão dinâmica, que corrige a recursão da gramática ao gerar os indivíduos, mas não funcionou bem, já que o artigo detalhava uma versão recursiva do algoritmo, que estourava a recursão máxima rapidamente.

Na lógica do meu algoritmo

O **genótipo** é um dicionário de não terminais e lista de índices de produções da gramática *n* vezes, indicando todas as produções que aquele não terminal fará

A **mutação** do genótipo ocorre mutando índices da lista aleatoriamente, escolhendo um número no intervalo de produções possíveis

O **crossover** se dá por uma máscara, que troca entre os filhos os genes dos pais

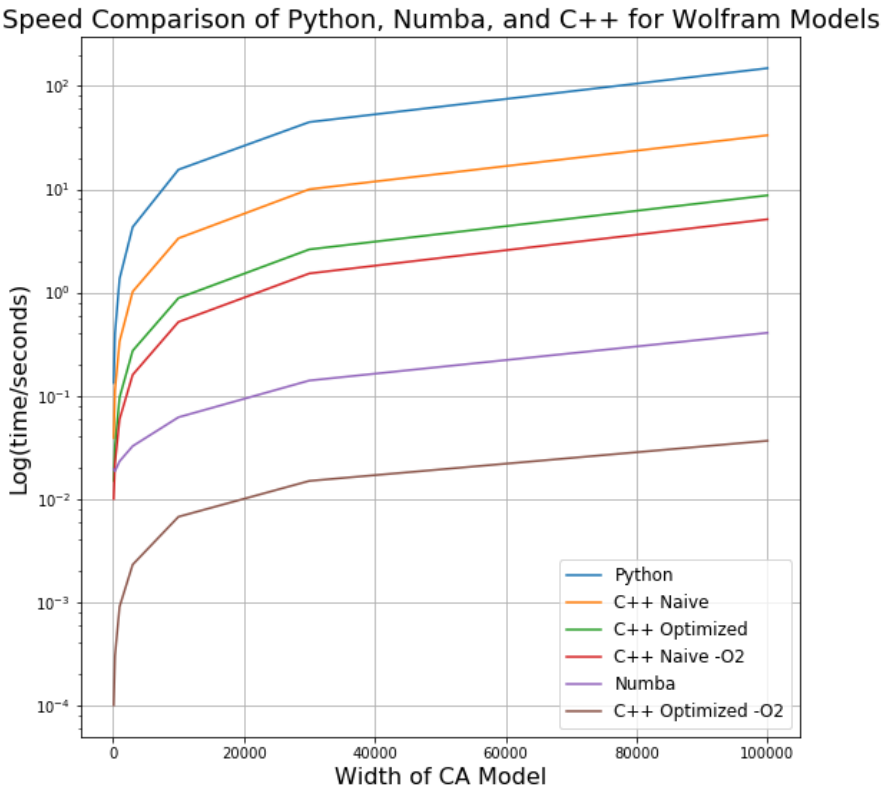
A **reprodução** se dá pela cópia direta do indivíduo, com probabilidade 1 - p_mut - p_cross

Implementação

A parte mais complicada desse trabalho é a conciliação de estruturas de dados abstratas e performance.

Quanto melhor a abstração, mais simples o trabalho, porém a performance sofre bastante.

Por isso, decidi implementar boa parte trabalho usando a biblioteca **Numba,** que compila código python para um dialeto de C, aumentando imensamente a velocidade do código.



Como o gráfico mostra, Numba é imensamente mais rápido que Python e C++, só perdendo para implementações inteligentes com flag de compilação O2, em tarefa de autômatos celulares

Tal decisão de implementação gastou um tempo absurdo, tanto pelos bugs quanto pelos comportamentos não documentados da biblioteca, que ainda é meio experimental, porém o ganho de performance foi justificável

Consegui fazer execuções extremamente rápidas, como a tabela abaixo, calculada com a biblioteca `time` do Python, mostra.

Repetições aleatórias	Gerações	Tamanho população	Profundidade gramática	Tempo de execução	Proporção de aumento dos parâmetros com o caso base	Aumento com o anterior
1	1	100	5	4.74 s	1x	1x
1	100	100	5	61.3s	100 x	10x
1	200	200	5	305.59 s	400x	4x
10	20	200	5	6000.53 s	4000x	10x

O tempo não aumenta de maneira constante com o aumento do tamanho do problema por decisões de implementação, como paralelismo e seleção de tamanho variável, além da influência do cache e a natureza estocástica do problema. Como o maior gargalo do algoritmo é a função de fitness, que precisa construir a árvore e computar o `eval` para todos os casos, ela limita a performance do algoritmo, e runs com fenótipos mais curtos performam melhor, algo totalmente aleatório.

Outro fator é são os cold starts, que fazem o numba recompilar o código e executá-lo, demorando bem mais na primeira execução.

Estrutura do código

Um notebook *jupyter* faz a maioria das operações relevantes, enquanto um arquivo scripts contém as implementações da maioria dos métodos. Tal decisão se deu pelo tamanho, que já excedia 1600 linhas, dificultando reiniciar o notebook toda hora. Além disso, o numba se beneficia de arquivos externos, já que compila previamente o código e, se ele for igual, não compila novamente, economizando tempo de execução entre restarts.

O código todo pode ser testado executando o arquivo start, desde que você tenha a versão correta do numba (acho que 0.57, a mais recente) e alguma sorte

Gramática

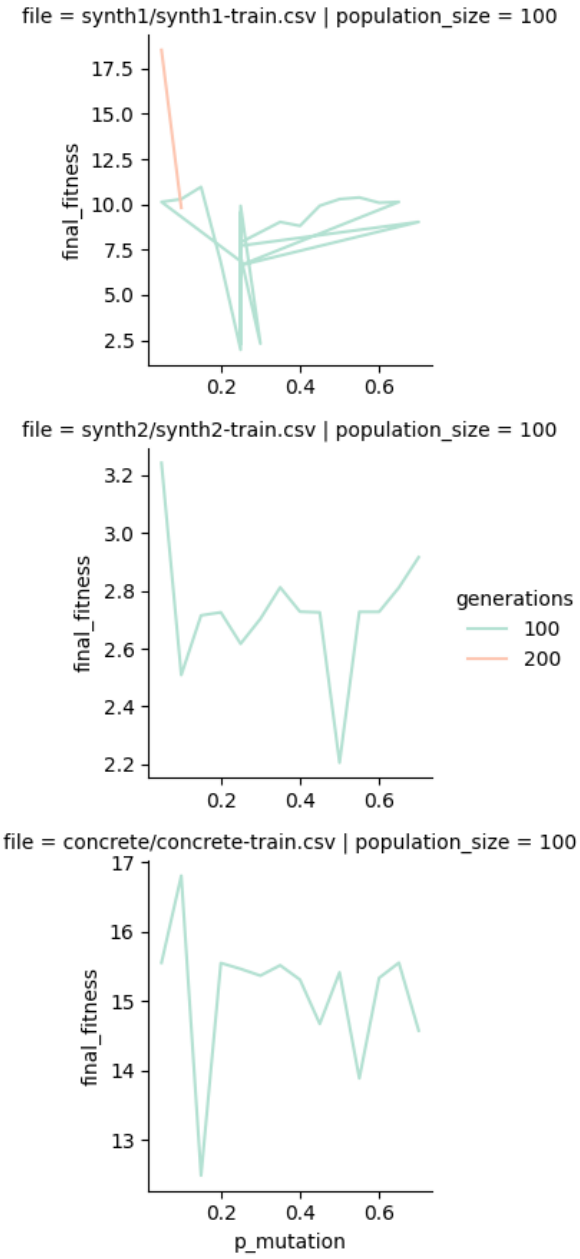
Criei uma gramática recursiva bnf simples, que é modificada de acordo com parâmetros de profundidade e número de variáveis, gerando um dicionário de não terminais e produções.

```
<start> ::= <expr>

<expr> ::= <number>
          | <var>
          | ( <op> <expr> <expr> )
          | ( <uop> <expr> )

<number> ::= <integer>

<integer> ::= ( <digit> )
            | ( <non-zero-digit> <digit> )

<digit> ::= 0 | <non-zero-digit>

<non-zero-digit> ::= 1
                   | 2
                   | 3
                   | 4
                   | 5
                   | 6
                   | 7
                   | 8
                   | 9
<op> ::= '+'
       | '-'
       | '*'
       | '/'

<uop> ::= 'log' | 'sin'

<var> ::= 'x'
```

A substituição de `<var>` em runtime só permite 10 variáveis, já que a lista de expansões permitidas é hard coded.

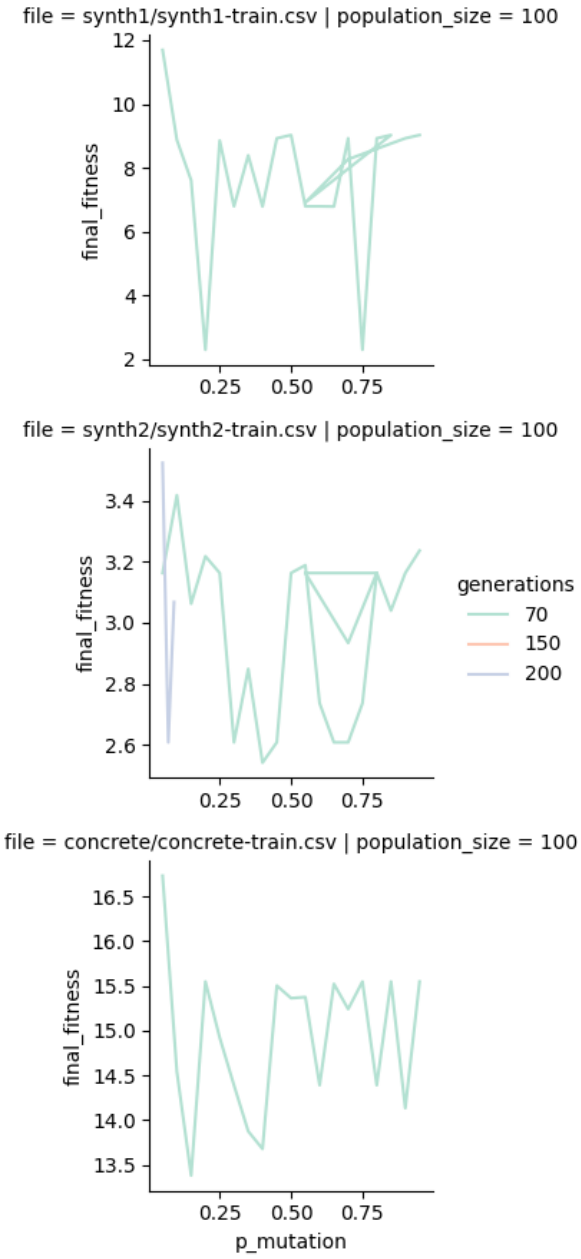
Em experimentações, com dados de execução nos arquivos `cacheSin.csv` e `cacheAbs.csv`, a melhor gramática foi a que combinava operações de logaritmo e seno, com a operação de `abs()` e `exp()` retiradas da gramática.

Rodei diversos experimentos para a decisão de utilizar seno ou não, e notei que sua utilização era benéfica. Fiz isso também para log, mas não gerei dados



uso de seno na gramática

A operação de valor absoluto é desnecessária, já que geralmente a solução convergirá para usar um produto por número negativo na maioria das vezes, já que poucos problemas realmente são modelados por uma função de valor absoluto.



métricas usando a função de abs

Por fim, percebi que a função exp era desnecessária também, já que tornava o y sensível demais a alterações em x, frequentemente convergindo para o limite superior ou inferior da imagem de exp (hard coded em 100000 e 0).

Estruturas de dados

Como Numba não suporta classes tão bem, implementei uma classe `NodeData`, que comporta *label* e uma lista com ids dos filhos, representando a árvore.

Existem estruturas de dados para a gramática, genótipo e fenótipo, ajustadas e tipadas devidamente com o Numba

A árvore da gramática, representando o fenótipo, é implementada por meio de uma tabela hash de ids, que sofre parse recursivo pela esquerda, consultando a tabela para recuperar as produções e nós filhos.

Para a **avaliação**, fiz uma função que recebe o genótipo e variáveis do problema e constrói o fenótipo. Com ele construído, vai recursivamente descendo a árvore até encontrar nós terminais, que são avaliados e retornados de acordo com as operações da gramática, retomando um único float no final.

Paralelismo

Fiz extenso uso de paralelismo, tanto com o `prange` quanto da biblioteca de concorrência do python, acelerando a conta de fitness, a operação mais custosa do problema, tendo speedup substancial.

Indivíduos iniciais

Eu calculo o limite superior de quantas produções cada token tem, que será n.

Cada gene no genótipo representa um não terminal e suas produções, então calcularei o n de cada gene. Para cada gene, criarei k índices de produções, de modo que qualquer gramática possível seja coberta. Cada índice é escolhido aleatoriamente frente as opções de produção desse termo.

Foi especificado tamanho 7, porém como meu genótipo não tem correlação com tamanho, dependendo da recursão da gramática, escolhi em variá-lo para representar o tamanho do indivíduo, de forma que o tamanho do indivíduo fosse 2 + profundidade de expressões, com o *max_depth* sendo escolhido como 5 para atender isso.

Seleção e pressão seletiva

Inicialmente fiz seleção por torneio, depois trocando para lexicase. O algoritmo seleciona a elite e os retira da população, para depois aplicar a lexicase. Na implementação clássica, a lexicase só retorna um indivíduo sobrevivente.

Como gostaríamos de selecionar um conjunto de indivíduos para sobreviver, teríamos que chamar várias vezes a seleção lexicase. Com isso, notei que seria mais eficiente retirar a restrição de retorno da lexicase, de modo que se vários indivíduos atendam um problema abaixo do erro epsilon e em todos os casos de teste, ela retorne todos.

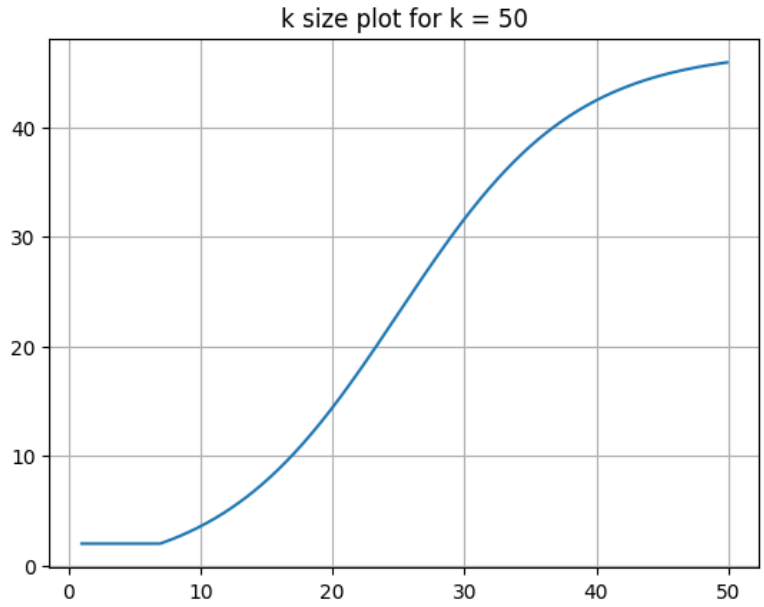
Criei uma função `lexicase_wrapper` que é responsável por chamar a lexicase várias vezes, de modo a construir um conjunto de sobreviventes com tamanho desejado. Um critério de parada fixo foi implementado, de modo que se o conjunto de sobreviventes não variar com novas chamadas da lexicase, o algoritmo parará e retornará menos indivíduos que o esperado.

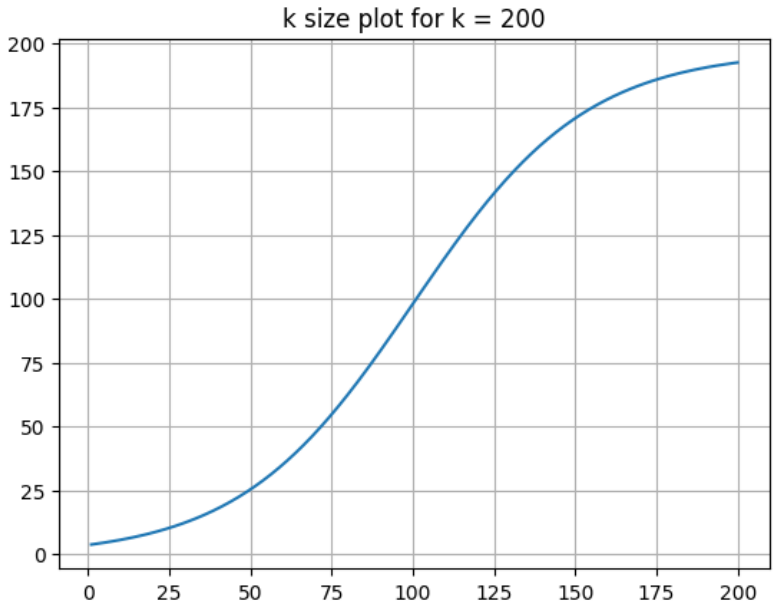
Determinar quantos indivíduos a seleção deve retornar foi um desafio, pois a pressão seletiva depende disso. Com isso, escolhi experimentalmente uma função para determinar o tamanho do “torneio”. Esse torneio funciona semelhantemente ao impacto de uma seleção de torneio k = 2 para 100 indivíduos: os 50 melhores são selecionados, logo, um aumento do k aumenta o *exploitation*, então é desejável que ele cresça não linearmente, senão rapidamente arriscaríamos mínimos locais.

Para isso, experimentei que a melhor função na minha implementação é uma sigmóide, que garante crescimento logístico, balanceando bem *exploration* e *exploitation.*

Para descobrir os coeficientes ideais, plotei graficamente com algumas funções lineares e variei os coeficientes até atingir o formato desejado.

$$f(x) = (k-2)\,\frac{1}{1+e^{-\frac{7.5}{k}\left(x-\frac{k}{2}\right)}} - 2$$





Mutação

A tarefa mais difícil foi lidar com o sistema de tipos do Numba, que não suporta cópia profunda, o que gerou muitos problemas. Criei um método estável o suficiente que copia manualmente o dicionário e é chamado frequentemente nas funções que fazem alterações inplace.

Fiz um tipo único de mutação intra gene, que dada uma probabilidade, percorre todas as listas de produções e, se a probabilidade for menor que o aleatório gerado, escolhe uma nova produção adequada para esse índice.

Para as mutações de genótipo, pensei em duas abordagens:

1. Rodar a mutação intra gene para todos os genes do genótipo
2. Rodar a mutação para um gene estocásticamente, ou seja, escolho um subconjunto do genótipo, rodo mutação intra gene em todo mundo e junto com o genótipo não modificado

Acabei optando pela implementação 1, que performou melhor, já que o grau adicional de liberdade com a introdução de um novo parâmetro experimental demandaria muito mais experimentação sem a garantia de melhora.

Crossover

O crossover foi definido como sendo a aplicação da máscara de crossover aleatória entre dois indivíduos sobreviventes, invés de dois indivíduos na população. Isso é problemático com seleção lexicase, então nesse caso fiz para que se existisse apenas um pai, ele o reproduzisse na população.

Gerações

Essa tarefa é feita na função `next_generation`. Em cada geração eu

- Calculo a fitness de todos os genótipos da população, aplico algum algoritmo de elitismo, selecionando os n melhores, e os retiro da população.
- Chamo alguma função de seleção, torneio ou lexicase
- Depois, em um loop, vou adicionando indivíduos na população, mutando, reproduzindo ou cruzando, até atingir o número de indivíduos na população desejado. Como existe uma chance de vir maior que o desejado, já que o cruzamento faz 2 indivíduos, removo os excedentes ao final.

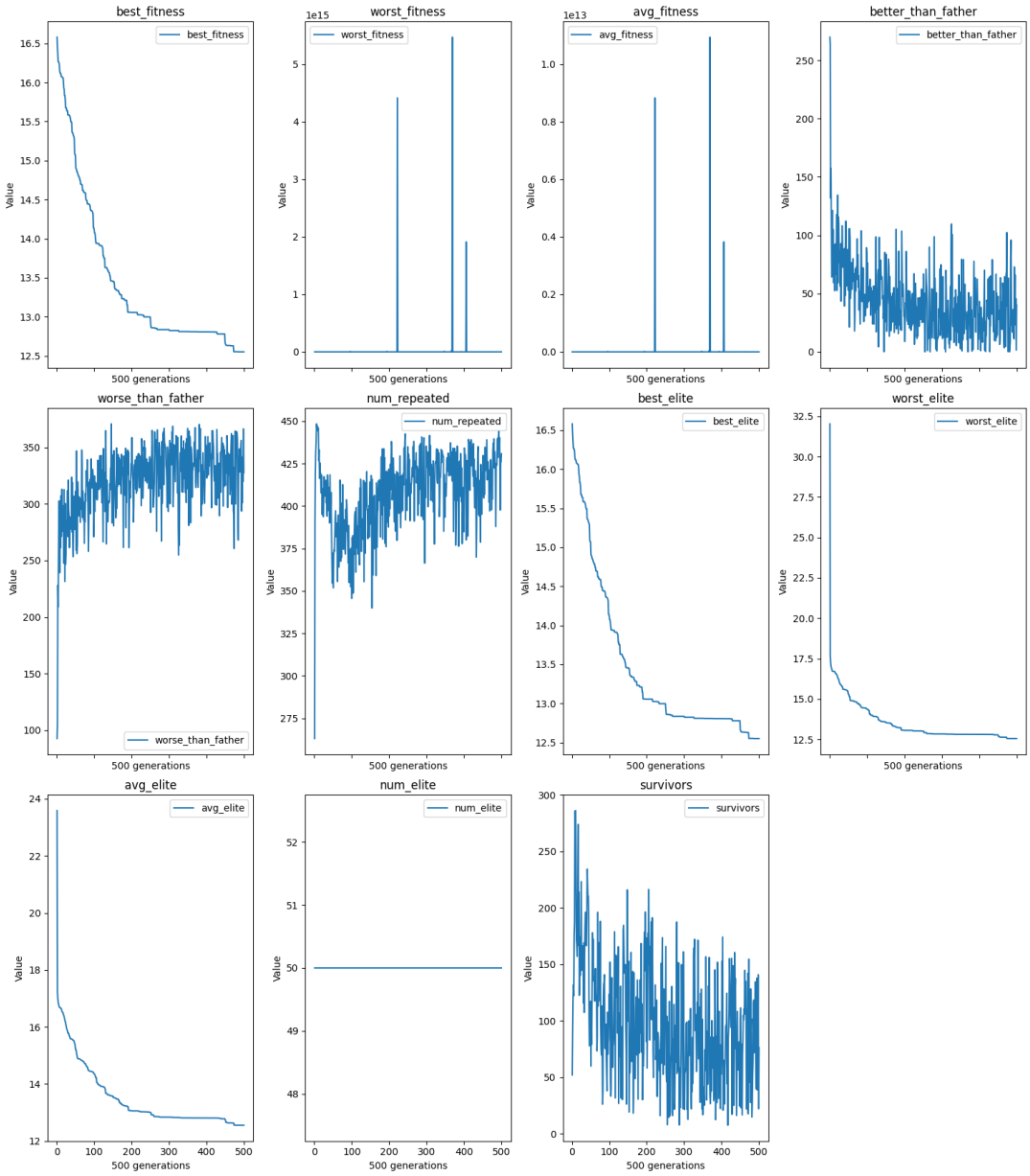
Essa função coleta todos os dados necessários em um array, que é utilizado depois para análises.

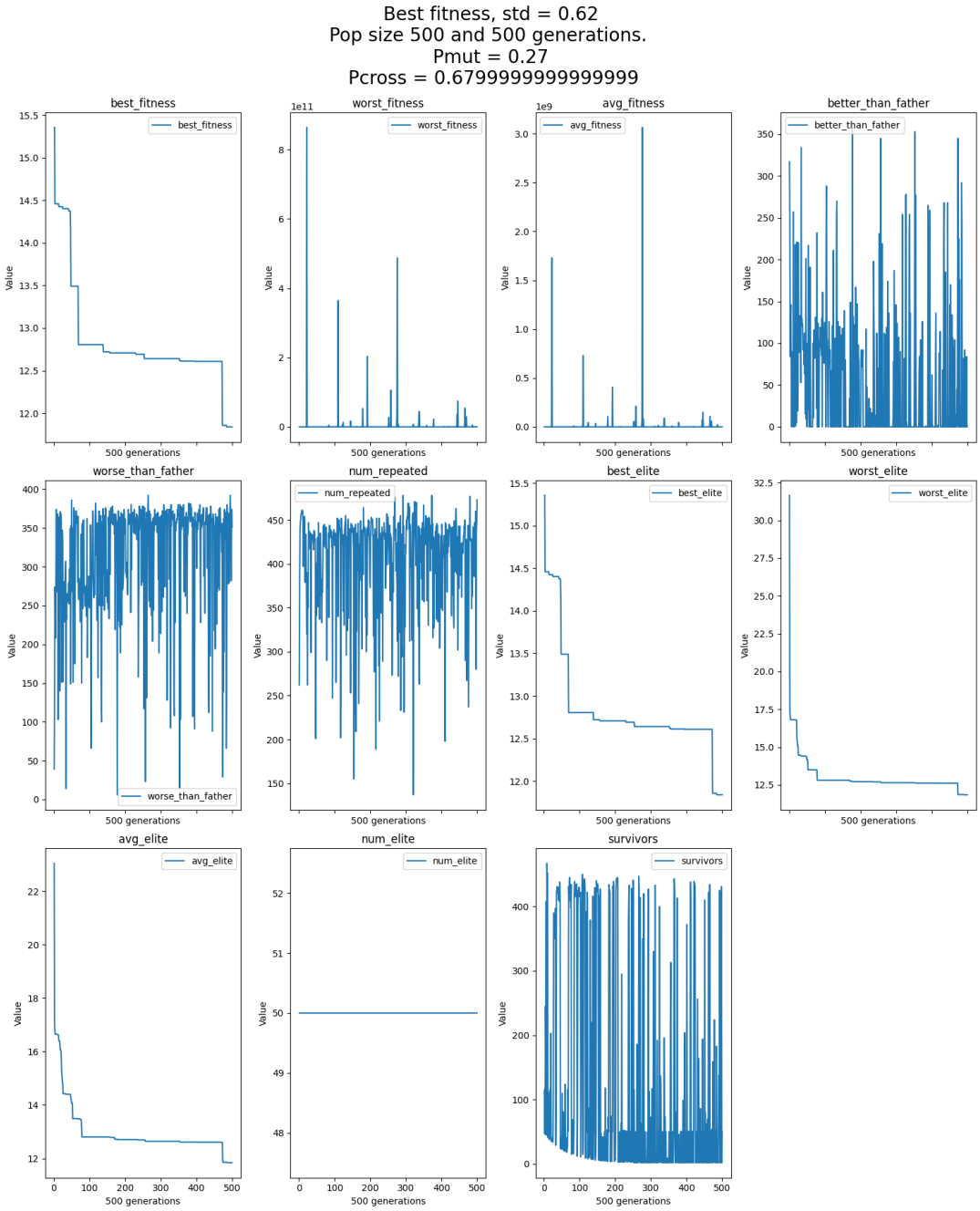
A contagem de indivíduos iguais é feita pelo fenótipo, de modo que indivíduos diferentes possam ser “iguais” nesse aspecto. Foi a melhor opção, já que utilizar hashes proporcionava colisão muito baixa e dificultava o diagnóstico de convergência de fenótipos.

Testes

Para testagem fiz uma função que recebia parâmetros do problema teste e executava n vezes, variando sementes aleatórias, e mostrava o desvio padrão, retornando os genótipos do melhor caso aleatório, chamada `run_random_variations`. Ela também mostra os gráficos da execução no melhor e caso médio para as variáveis do problema. Optei por fazer 10 variações aleatórias, já que o tempo de execução estava alto demais para fazer 30.

Segue uma saída dos gráficos para testes da base concreta


Mean values for 10 runs, std = 0.62
Pop size 500 and 500 generations.
Pmut = 0.27
Pcross = 0.6799999999999999
best_fitness
worst_fitness
avg_fitness
better_than_father
worse_than_father
num_repeated
best_elite
worst_elite
avg_elite
num_elite
survivors
Value
500 generations



Cache

Para evitar computações custosas se repetindo, criei uma estratégia de cache, em que se um experimento já foi rodado com aqueles parâmetros, ele era pulado. Isso me gerou bastante problema, com um bug originado de um erro de digitação salvando a variável incorreta em um campo inutilizando dias e dias de experimentos.

Comparação com a base de teste

Escrevi uma função de comparação, consumindo os n melhores genótipos, adquiridos na experimentação, e avaliando-os frente aos casos de teste de outro arquivo, para depois plottar um bar plot do fitness para cada genótipo.

Em uma regressão simbólica boa, é esperado que a fitness seja muito baixa em vários casos, já que esses genótipos representam a última população do algoritmo.

Experimentos

Fiz várias runs variando o tamanho da população, profundidade da gramática, número de gerações e o tipo de mutação utilizado. Para obter uma noção melhor dos valores, fiz o método de execução aleatória retornar os desvios padrões das variáveis entre runs com sementes diferentes, de modo que o problema se tornou a minimização de desvio padrões após a convergência do algoritmo.

Conclui que os melhores parâmetros para experimentação era população e número de gerações fixados em 200 e profundidade da gramática fixada em 5, então variei de 5 em 5% a mutação. Cada experimento demorava em torno de 1h, e os repeti para diferentes gramáticas.

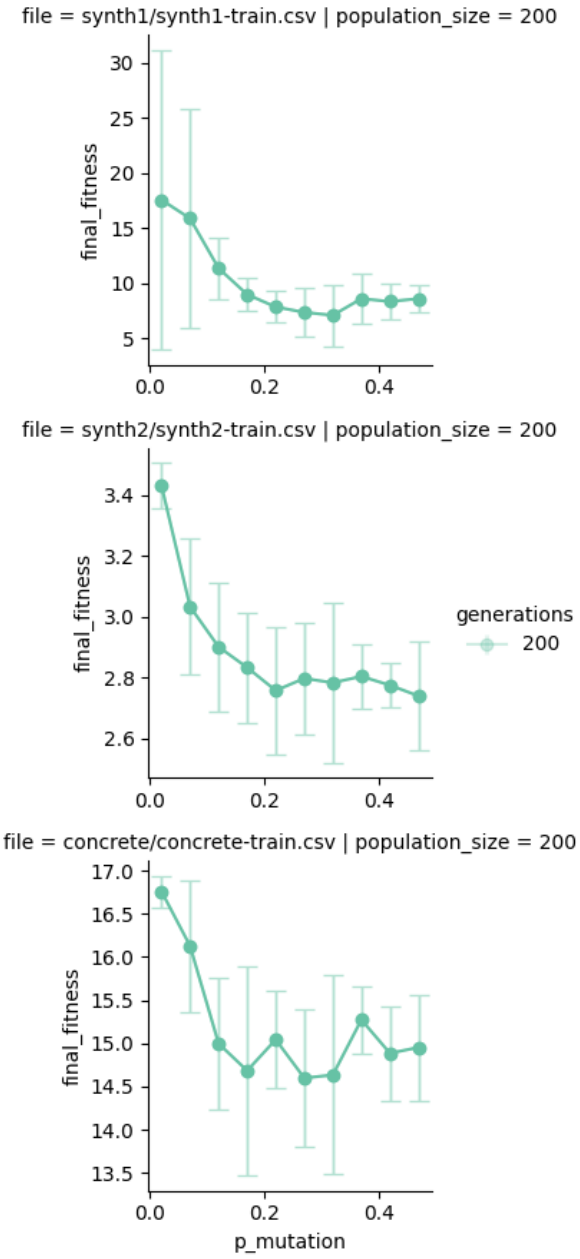
Runs com 500 gerações e população de 500 genótipos foram a com melhor desempenho, embora seu tempo de execução tornasse inviável: aproximadamente 8h por caso. Testei-as para o caso concreto.

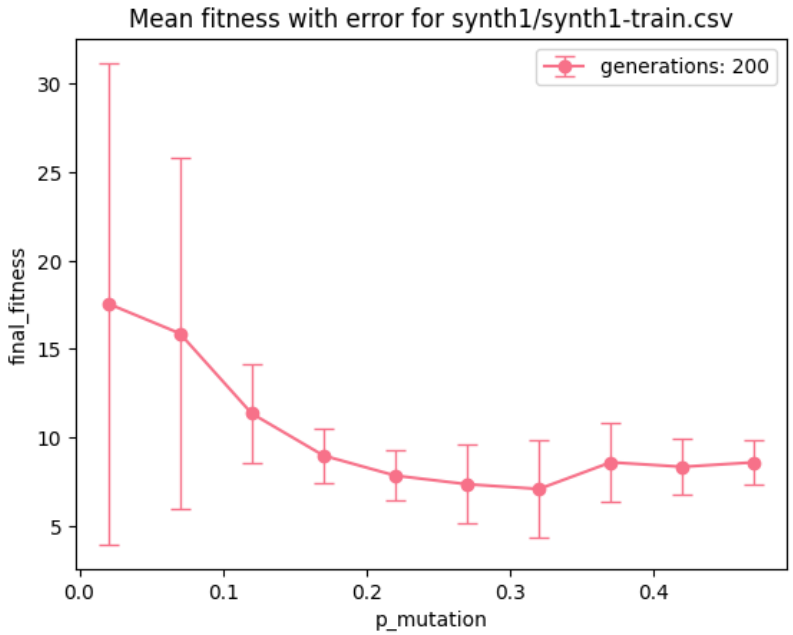
Como a máquina que tive é bem limitada, não fui muito além do básico das métricas, sendo talvez melhor aumentar o número de variações aleatórias e experimentar com mais tamanhos de população.

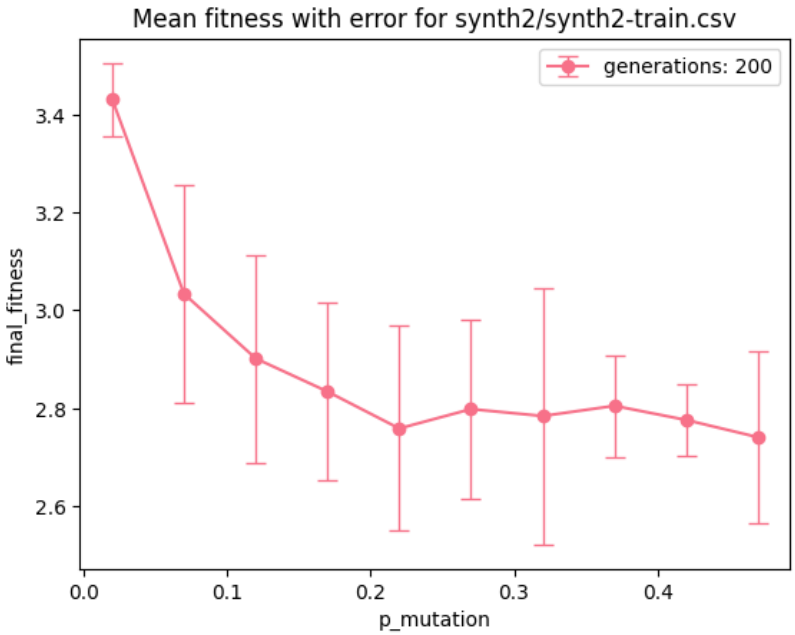
Além disso, elitismo foi outro fator que não consegui retirar, com as soluções ficando piores sem ele. Pelo menos uma vez na run o algoritmo explodia para um fitness horrível, e o elitismo ajudava a guiar de volta para a solução ideal do problema, sem perder o espaço de buscas

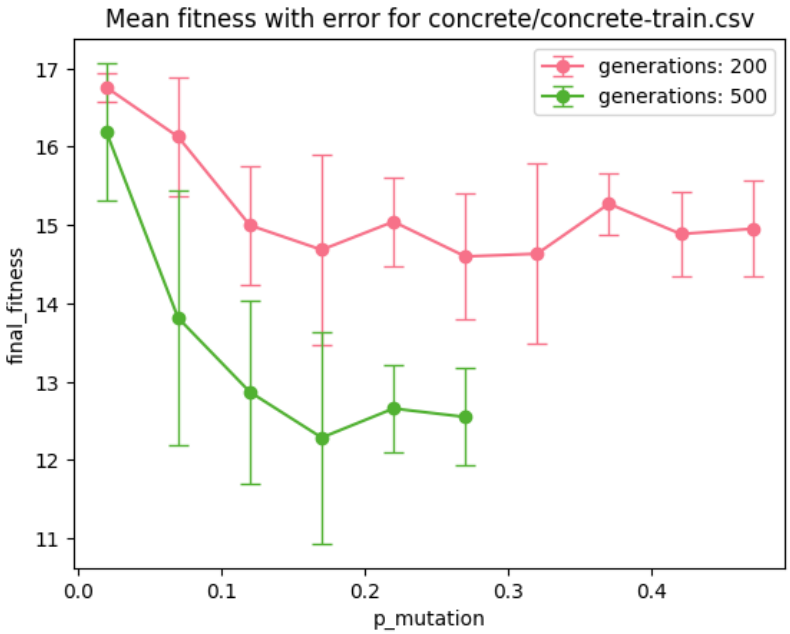
Para descobrir o resultado das minhas experimentações, criei outro notebook específico para análise.

Seguem meus experimentos com população 200 e 200 gerações, com 10% da população sendo elitizada e lexicase variável.


file = synth1/synth1-train.csv | population_size = 200
final_fitness
30
25
20
15
10
5
0.0
0.2
0.4
file = synth2/synth2-train.csv | population_size = 200
final_fitness
3.4
3.2
3.0
2.8
2.6
0.0
0.2
0.4
generations
200
file = concrete/concrete-train.csv | population_size = 200
final_fitness
17.0
16.5
16.0
15.5
15.0
14.5
14.0
13.5
0.0
0.2
0.4
p_mutation


Mean fitness with error for synth1/synth1-train.csv
generations: 200
final_fitness
p_mutation
30
25
20
15
10
5
0.0
0.1
0.2
0.3
0.4


Mean fitness with error for synth2/synth2-train.csv
generations: 200
final_fitness
p_mutation
3.4
3.2
3.0
2.8
2.6
0.0
0.1
0.2
0.3
0.4



no caso de 500 gerações, também aumentei a população para 500

Conclusões

Foi muito interessante fazer esse trabalho, principalmente sobre como implementar os algoritmos de maneira inteligente e eficiente. Tive bastante curiosidade de variar os parâmetros e chegar no melhor valor possível

Bibliografia

Lourenço, Nuno & Pereira, Francisco & Costa, Ernesto. (2016). Unveiling the properties of structured grammatical evolution. Genetic Programming and Evolvable Machines. 17. 10.1007/s10710-015-9262-4.

Lourenço, Nuno & Assunção, Filipe & Pereira, Francisco & Costa, Ernesto & Machado, Penousal. (2018). Structured grammatical evolution: A dynamic approach. 10.1007/978-3-319-78717-6_6.