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.

Em seguida, criamos o **genótipo**, que é um dicionário de não terminais e lista de índices de produções da gramática.

A mutação do genótipo ocorre de três formas diferentes, acabei optando pela da literatura (mutar índices da lista aleatoriamente)

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 pyton para C, aumentando imensamente a performance.

Tal decisão gastou um tempo absurdo, tanto pelos bugs quanto pelos comportamentos não documentados da biblioteca (falarei mais depois), porém o ganho de performance foi justificável:

Consegui fazer uma run de 10 populações de tamanho 100, com 20 gerações, em cerca de 5 minutos (menos se eu tirasse os prints do código)

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> )
            | ( <non-zero-digit> <digit> <digit> )

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

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

<uop> ::= 'abs'

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

O código só permite 10 variáveis, já que a lista de expansões permitidas é hard coded.

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 (outra fonte de problemas)

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.

Indivíduos iniciais

Basicamente 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. Em cada gene, criarei n 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 7 para atender isso.

Seleção

Inicialmente fiz seleção por torneio, depois trocando para lexicase. O algoritmo seleciona a elite e os retira da população, para depois aplicar a lexicase. Geralmente a lexicase só retorna um indivíduo sobrevivente (como o caso da minha implementação)

Mutação e crossover

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 três abordagems:

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
3. Escolher aleatoriamente mutar ou não genótipos inteiros em uma lista de genótipos (caso 1 aplicado para n genótipos). Acabei não usando

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 mutasse o filho invés de crossover.

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, entre torneio e 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 fim.

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 iniciais

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. Em seguida, uma função de comparação era chamada, consumindo os n genótipos 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.

Testes de parâmetros

Executei a função de teste variando parâmetros de mutação e crossover, analisando os gráficos de retorno, para os 3 datasets de teste e treino, de modo a verificar o impacto da variação desses parâmetros.

Seleção lexicase

Tive bastante trabalho mas adaptei-a para funcionar com o numba, de modo que um subconjunto de genótipos fosse escolhido, fitness calculada e os indivíduos fora do epsilon permitido seriam removidos, o que se repete até estourar o número de casos de teste ou ter apenas um indivíduo restante.

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.

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 um método que printava os desvios padrões das variáveis entre runs com sementes diferentes, de modo que o problema se tornasse a minimização de desvio padrões após a convergência do algoritmo.

Como o tempo e máquinas que tive foram bem limitados, não fui muito além do básico das métricas, mas notei que em minha implementação mutação tinha um papel muito importante, já que o crossover frequentemente ficava preso em indivíduos com fenótipos idênticos.

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

Conclusões

Foi muito interessante fazer esse trabalho, gostaria de ter conseguido experimentar mais com variações dos parâmetros nas bases de dados, o que foi inviável pelo run time do meu algoritmo sem a ajuda do numba devido à seleção lexicase. Se possível ter mais prazo, vou quebrar a cabeça para paralelizar e rodar mais com os outros datasets.

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.