Universidade Federal de Minas Gerais

Computação Natural

Programação Genética

Ingrid Rosselis Sant'Ana da Cunha

Outubro 2017

Introdução

O primeiro trabalho prático da disciplina Computação Natural tem como objetivo a implementação de um algoritmo de regressão simbólica utilizando programação genética (GP). Este método estima uma expressão desconhecida baseando-se na aproximação por pontos conhecidos. Mais especificamente, o método de regressão simbólica procura no espaço das expressões matemáticas um modelo que melhor se encaixe nos pontos dados em um *dataset*. O objetivo final é conseguir uma expressão que atenda os quesitos de precisão e simplicidade para qualquer *dataset* dado da mesma expressão.

Um dos melhores e mais conhecidos modos de implementar um algoritmo de regressão simbólica é através da programação genética. A GP permite que o método se diferencie dos demais por não possuir viés inicial, isto é, o "modelo" inicial para o método é feito de forma aleatória, sem ser afetado por conhecimento insuficiente do utilizador em relação ao problema ou por pré-julgamentos errôneos com relação à forma da expressão. O método também tem a vantagem de permitir ao utilizador uma escolha entre o melhor *trade-off* de desempenho e precisão.

Implementação

Como rodar o trabalho prático e alterar seus parâmetros

O programa criado para a resolução do trabalho prático foi feito em *Python 3*. Para sua execução, o diretório `../tp1/src` deve ser acessado via terminal e o comando `make` deve ser executado (pode-se também executar alternativamente com o comando `python3 gp_main.py`, porém não é recomendado pois o *Makefile* executa a limpeza de arquivos objetos e temporários). Recomenda-se também que as pastas internas não sejam alteradas, para que não comprometa a leitura dos *datasets* e a gravação dos *logs*, salvo os casos onde os diretórios forem alterados no código.

Quando executado, o programa imprime o número da execução e da geração sendo rodadas no momento, para fins de controle, e ao final o *log* respectivo estará na pasta `../tp1/logs`.

Todos os parâmetros, constantes e variáveis estão localizados no arquivo `parameters.py`. Os parâmetros que podem ser modificados neste trabalho sem alterar o funcionamento do código são os seguintes:

- `CSV_NAME`: nome do arquivo *dataset* a ser usado;
- `CSV_DIR`: diretório do *dataset*;
- `LOG_ADD`: parte complementar ao nome do *log*;
- `LOG_DIR`: diretório onde estará o arquivo de saída;
- `EXT_LOG`: extensão do arquivo de saída;
- `TYPE`: tipo de dado que está sendo utilizado (podendo ser "*train*" ou "*test*");
- `EXEC`: número de execuções que o algoritmo será rodado;
- `MAX_POPULATION`: tamanho máximo da população em qualquer geração;
- `INITIAL_POPULATION`: tamanho da população gerada aleatoriamente;
- `MAX_GENERATION`: número máximo de gerações em cada execução;
- `PROB_CROSSOVER`: probabilidade de cruzamento entre dois indivíduos;
- `TOURNAMENT_SIZE`: número de indivíduos a participar do torneio por vez;
- `MIN_OPTIMAL_SOL`: solução ótima para encerrar a execução de um algoritmo (apenas usado para tipo "*test*");
- `ELITISM`: quantos indivíduos ótimos serão passados para a próxima geração;
- `MAX_DEPTH_TREE`: tamanho máximo do indivíduo.

Os seguintes parâmetros são dependentes de outros parâmteros/atributos do programa e não devem ser alterados:

- `LOG_NAME`: é, por definição, o `CSV_NAME` concatenado com `LOG_ADD`;
- `EXT_CSV`: para facilitar a implementação, o programa trata exclusivamente de entradas `.csv`;
- `PROB_MUTATION`: a probabilidade de mutação foi fixada no programa como sendo o complemento da probabilidade de crossover, isto é, $1 - prob\_crossover$;
- `MAX_OPERATORS`: número de operadores só pode ser alterado se o usuário estiver certo de quais operadores serão utilizados, dada a lista de operadores definidos;
- `MAX_VAR`: o número de variáveis utilizadas para gerar indivíduos depende do número de variáveis existentes no *dataset* passado.

Detalhes de implementação

Como a maioria dos algoritmos de programação genética, o indivíduo neste trabalho é representado por uma árvore binária. Cada indivíduo é composto por uma árvore que o representa, a profundidade desta árvore e sua *fitness*. Cada nodo da árvore possui um tipo, uma chave, uma profundidade e dois nodos filhos (esquerda e direita). Segue abaixo uma descrição dos atributos:

- Tipo: representa o atributo que existe naquele nodo. Pode ser operador, variável ou constante;
- Chave: varia de acordo com o tipo. Mais especificamente:
  - Operador: pode ser $+, -, *, /$ ou log;
  - Variável: pode ser x_i, tal que $i = 1, .., MAX\_VAR$;
  - Constante: um número entre o `MIN_VALUE_CONSTANT` e o `MAX_VALUE_CONSTANT` com número de casa decimais depois da vírgula dado por `PRECISION_DIGITS`.
- Profundidade: o quão próximo das folhas está o nodo. Esta informação é essencial para decidir de qual tipo um nodo pode ser e onde e como o indivíduo pode ser mutado/cruzado.
- Filhos: os filhos são a continuação de uma árvore a partir do nodo pai. Para as contantes e variáveis, os nodos pais não possuem filhos, e para os operadores, os nodos pais podem ter 1 ou 2 filhos dependendo da operação indicada no nodo pai (1 filho se o operador é log, 2 filhos para os demais).

O algoritmo foi dividido em grandes partes, sendo elas: geração de uma população inicial, escolha do indivíduo elitista, controle do tamanho da população, seleção de indivíduos, cruzamento/reprodução, mutação dos novos indivíduos e, por fim, a troca de população. Os cálculos da *fitness* e da profundidade de uma árvore são feitos logo após a modificação desta, ou seja, quando uma nova árvore é gerada, independente se o processo é de geração aleatória, cruzamento ou mutação.

A primeira parte do algoritmo é a criação de uma população inicial de tamanho fixo. Nesta parte, foram feitas uma série de escolhas estocásticas referentes ao diversos tipos de árvores que podem ser geradas. Para gerar um novo indivíduo, o algoritmo seleciona os tipos do nodo, a chave que o nodo representará e marca em qual profundidade o novo nodo se encontra. Após esses passos, o novo nodo é gerado e seus filhos são definidos de acordo com o tipo e a chave.

A escolha do tamanho da árvore deu-se por uma série de testes entre os tamanhos 4 e 7. A ideia era que ela não ficasse extremamente grande - o que levaria a uma expressão demasiadamente complexa - e que conseguisse representar boas soluções para os *datasets*. No fim, foi escolhido com 5 o tamanho padrão da árvore neste programa. Para o crescimento da árvore, foi utilizado o método *grow*.

Após gerar a população inicial, o algoritmo passa a rodar uma sequência de passos referentes a uma única geração, que constitui a evolução de uma população. Primeiro, o elemento de melhor *fitness* é copiado diretamente para a nova população. A seguir, o tamanho da população atual é conferido de acordo com a `MAX_POPULATION`. Se estiver acima do valor definido, os piores elementos são retirados até que se chegue à quantidade desejada. Depois os elementos

são retirados da população atual por seleção até que esta fique vazia. Se a população possuir tamanho maior ou igual a 2, duas seleções por torneio são feitas e os indivíduos ganhadores são retirados da população e são colocados para cruzar. Caso eles cruzem, seus filhos vão para a lista que será mutada logo a seguir. Caso contrário, os pais concorrem em um torneio local e o melhor sofrerá um tipo de reprodução. Se o número de indivíduos na população for ímpar, o último indivíduo restante também sofrerá a reprodução. Por fim, os indivíduos gerados por *crossover* e aqueles que foram reproduzidos tem uma probabilidade `PROB_MUTATION` de sofrerem mutação. A seguir os novos indivíduos são colocados na nova população. Nesta etapa, duas decisões de projeto foram tomadas:

1. O controle do tamanho da população é feito com a população antiga antes do algoritmo de seleção ser executado, e o pior indivíduo é retirado. O motivo dessa escolha foi porque em vários testes não foi comprovada uma diferença relevante entre a retirada de um indivíduo aleatório ou do pior indivíduo, e, para não correr o risco de tirar os indivíduos com *fitness* boas, foi definido retirar sempre o de maior *fitness*. Essa escolha tira a possibilidade de que o pior elemento gere um novo elemento ótimo, mas é mais segura em manter elementos bons na população;

2. Foi definido que os indivíduos que sofressem reprodução seriam colocados juntos com os novos indivíduos na lista onde cada elemento tem uma probabilidade de mutação. O motivo dessa escolha é que foi comprovado que faltava diversidade na população e estava convergindo rápido demais. Decidiu-se, assim, mutar esses elementos ao invés de só copiá-los na tentativa de gerar indivíduos diferentes/melhores.

Para os *datasets* de treino, o algoritmo roda todas as gerações especificadas como parâmteros. Já para os de teste, o algoritmo para na geração onde o indivíduo de melhor *fitness* se encaixa dentro do `MIN_OPTIMAL_SOL`.

Depois que a nova população é gerada, esta substitui a população antiga e o algoritmo se repete para as próximas gerações. O programa inteiro executa um número de vezes definido no parâmetro `EXEC`, pois devido ao fato de utilizar métodos estocásticos, não se pode basear os dados obtidos em apenas uma execução.

Experimentos

Para a avaliação dos melhores parâmetros para cada *dataset* foram definidos dois tipos de conjuntos de valores. O primeiro para os *datasets* **Keijzer 7** e **Keijzer 10** e o segundo para o **House**.

Os testes levaram em conta as medidas de melhor, média e pior *fitness* médias por execução e por geração. As médias por execução serviram para escolher os parâmetros e as médias por geração serviram para mostrar como a *fitness* do melhor indivíduo evolui com o passar das gerações. A execução dos testes seguiu a mesma ordem para todas as entradas de dados, sendo ela:

1. Testar casos base e definir tamanho de população e geração máximas;

2. Testar o melhor parâmetro de `PROB_CROSSOVER` para a população e geração escolhidas;

3. Testar o melhor `TOURNAMENT_SIZE`;

4. Conferir a diferença que o elitismo faz na solução do problema;

Para os **Keijzers**, foram utilizados vários testes para fazer a avaliação dos melhores parâmetros e para conhecer mais sobre o problema, mas os seguintes valores foram usados para os testes formais que geram os dados desta seção:

- População: 100, 200, 300;
- Geração: 100, 200, 300;
- Probabilidade de *crossover*: 0.9, 0.6;
- Tamanho do torneio: 2, 3, 5;
- Elitismo: 0, 1;
- Tamanho da população inicial: 50.

Os testes bases foram feitos de acordo com a Tabela 1.

População	Geração	Probabilidade de *crossover*	Torneio	Elitismo
100	100	0.9	2	1
100	200	0.9	2	1
100	300	0.9	2	1
200	100	0.9	2	1
200	200	0.9	2	1
200	300	0.9	2	1
300	100	0.9	2	1
300	200	0.9	2	1
300	300	0.9	2	1

Tabela 1: Esquema de testes dos parâmetros *população* e *geração* para os **Keijzers**

Para o **House**, foram formalizados os seguintes valores:

- População: 10, 15, 20, 30;
- Geração: 20, 50, 100, 300;
- Probabilidade de *crossover*: 0.9, 0.6, 0.95;
- Tamanho do torneio: 2, 3, 5;
- Elitismo: 0, 1;
- Tamanho da população inicial: 5.

Os testes bases foram feitos de acordo com a Tabela 2.

População	Geração	Probabilidade de *crossover*	Torneio	Elitismo
10	20	0.9	2	1
10	50	0.9	2	1
10	100	0.9	2	1
20	20	0.9	2	1
20	50	0.9	2	1
20	100	0.9	2	1
30	20	0.9	2	1
30	50	0.9	2	1
30	100	0.9	2	1
15	300	0.95	2	1

Tabela 2: Esquema de testes dos parâmetros *população* e *geração* para o **House**

Keijzer 7

Os testes base para este *dataset* resultaram nos seguintes valores de *fitness* média por execução:

População	Geração	Melhor	Média	Pior
100	100	0.31430	$5.03665 \cdot 10^6$	$1.91118 \cdot 10^8$
100	200	0.31461	$1.93185 \cdot 10^7$	$7.29189 \cdot 10^8$
100	300	0.31377	$2.83257 \cdot 10^6$	$9.38932 \cdot 10^7$
200	100	0.31571	$1.70038 \cdot 10^6$	$7.41317 \cdot 10^7$
200	200	0.31175	$2.24938 \cdot 10^8$	$9.61588 \cdot 10^9$
200	300	0.31059	$6.39649 \cdot 10^7$	$2.61495 \cdot 10^9$
300	100	0.31286	$3.35275 \cdot 10^7$	$1.33758 \cdot 10^9$
300	200	0.31175	$4.77740 \cdot 10^6$	$2.05992 \cdot 10^8$
300	300	0.31305	$5.82649 \cdot 10^5$	$2.12607 \cdot 10^7$

Tabela 3: Testes dos parâmetros *população* e *geração*

Considerando-se a melhor *fitness* média, foram escolhidos como melhores parâmetros a população máxima em 200 indivíduos e 300 como geração máxima. A avaliação destes dados comprovou também que este *dataset* funciona pior com um número menor de gerações e necessita de um número médio ou grande de indivíduos na população máxima para apresentar os melhores resultados.

A seguir foram feitos os testes com a probabilidade de *crossover*. Para estes, foi obtido os seguintes resultados:

Probabilidade de *crossover*	Melhor	Média	Pior
0.6	0.31165	$6.15763 \cdot 10^5$	$8.55392 \cdot 10^6$
0.9	0.31188	$2.35275 \cdot 10^5$	$1.02171 \cdot 10^7$

Tabela 4: Testes do parâmetro *probabilidade de crossover*

Apesar das melhores *fitness* médias terem resultado em valores muito próximos, foi escolhido manter a probabilidade em 0.9 por causa da mutação que ficaria com um valor muito alto caso a probabilidade de 0.6 para *crossover* fosse escolhida (lembrando que a probabilidade de mutação foi definida como o complemento da probabilidade de *crossover*). No entanto, os valores mostram que muito provavelmente a diversidade na população e o tamanho desta não eram suficientes para que a diferença entre os dois parâmetros fosse grande.

O próximo parâmetro a ser testado foi o tamanho do torneio.

Tamanho do torneio	Melhor	Média	Pior
2	0.30920	$7.82821 \cdot 10^6$	$2.85328 \cdot 10^8$
3	0.31479	$3.55650 \cdot 10^6$	$1.27991 \cdot 10^8$
5	0.31164	$3.48460 \cdot 10^7$	$1.39173 \cdot 10^9$

Tabela 5: Testes do parâmetro *tamanho do torneio*

O tamanho do torneio não fez muita diferença na melhor *fitness* média pois o tamanho da população máxima de 200 indivíduos foi grande o suficiente para que as três opções se mantivessem paréas.

Por último, conferiu-se a diferença de soluções para um algoritmo usando elitismo e outro sem.

Elitismo	Melhor	Média	Pior
0	$3.39918 \cdot 10^5$	$1.80898 \cdot 10^6$	$2.46134 \cdot 10^7$
1	0.32590	$6.55158 \cdot 10^10$	$2.86775 \cdot 10^12$

Tabela 6: Testes do parâmetro *elitismo*

Aqui, a escolha do parâmetro fez uma grande diferença. Pode-se dizer que sem o elitismo a busca se tornou aleatória e a solução não alcançou um resultado próximo ao desejável.

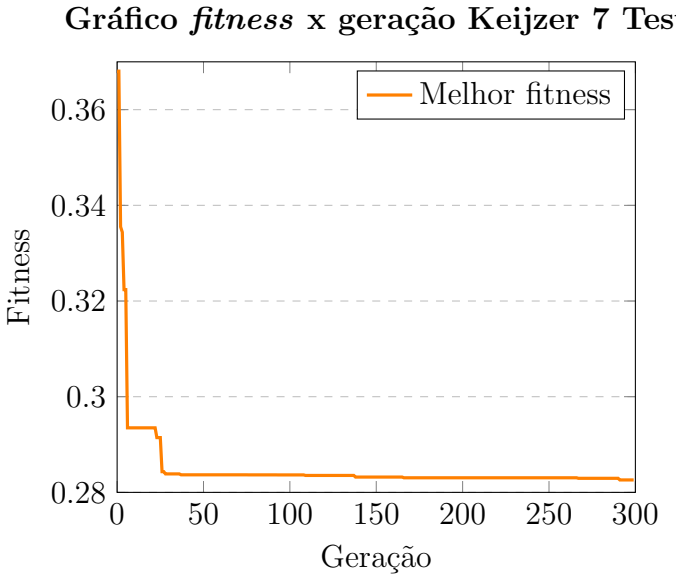
Por último, foi executado o *dataset* de teste para o **Keijzer 7**. Um resumo dos parâmetros escolhidos, as *fitness* médias e o gráfico segue abaixo:

População	Geração	*Crossover*	Mutação	Torneio	Elitismo	População Inicial
200	300	0.9	0.1	2	1	50

Tabela 7: Resumo dos parâmetros utilizados

Melhor	Média	Pior
0.28512	$7.09 \cdot 10^2$	$2.9054 \cdot 10^4$

Tabela 8: Resultado de *fitness* média para o *dataset* de teste



Pode-se perceber pelo gráfico que a pressão seletiva estava muito alta desde o início, que a população possuía pouca diversidade e que por isso o espaço de busca não foi eficientemente explorado, porque o gráfico convergiu para um valor bem próximo do resultado final antes da geração 50. Muito provavelmente a taxa de mutação deveria ter sido aumentada quando o algoritmo começou a convergir para que uma nova melhor solução pudesse ser encontrada.

Keijzer 10

Os testes base para o **Keijzer 10** resultaram nos seguintes valores de *fitness* média por execução:

População	Geração	Melhor	Média	Pior
100	100	0.15798	$3.09 \cdot 10^2$	$9.944 \cdot 10^3$
100	200	0.17389	$5.21 \cdot 10^2$	$12.561 \cdot 10^3$
100	300	0.18570	$6.480 \cdot 10^3$	$2.30967 \cdot 10^5$
200	100	0.18378	$9.788 \cdot 10^3$	$2.70091 \cdot 10^5$
200	200	0.15716	$1.500 \cdot 10^3$	$3.9576 \cdot 10^4$
200	300	0.14846	$4.8724 \cdot 10^4$	$2.00266 \cdot 10^6$
300	100	0.18523	$7.67 \cdot 10^2$	$2.0003 \cdot 10^4$
300	200	0.16290	$5.181 \cdot 10^3$	$1.33642 \cdot 10^5$
300	300	0.16668	$6.208 \cdot 10^3$	$2.00384 \cdot 10^5$

Tabela 9: Testes dos parâmetros *população* e *geração*

Para estes casos base, houve uma maior diferença entre os valores da *fitness* média. Apesar do erro das medidas de *fitness* média e pior terem ficado maiores que a dos outros testes, foi escolhido novamente os valores de população máxima igual a 200 indivíduos e geração máxima igual a 300. Pode-se reparar que os melhores resultados se encontraram em casos onde a diferença entre o máximo da população e o máximo de geração não era grande, dando a entender que este *dataset* funciona melhor para parâmetros médios e mais uniformes.

O próximo passo é o teste com a probabilidade de *crossover*.

Probabilidade de *crossover*	Melhor	Média	Pior
0.6	0.16184	$1.05 \cdot 10^2$	$7.62 \cdot 10^2$
0.9	0.15118	$1.772 \cdot 10^3$	$3.9758 \cdot 10^4$

Tabela 10: Testes do parâmetro *probabilidade de crossover*

Para o **Keijzer 10** fica clara a diferença entre as probabilidades de *crossover* altas e baixas. A probabilidade em 0.6 denota a maior aproximação de uma busca aleatória, pois sua convergência é menos precisa e é provavelmente mais lenta que com a probabilidade alta. É curioso notar também que a probabilidade mais baixa torna as diferença de valores menos discrepantes. Foi escolhido a probabilidade 0.9 pela diminuição da *fitness* média do melhor indivíduo.

A seguir, foi testado as diferentes possibilidades para o tamanho do torneio.

Tamanho do torneio	Melhor	Média	Pior
2	0.15707	$3.77 \cdot 10^2$	$9.106 \cdot 10^3$
3	0.15110	$2.26687 \cdot 10^5$	$6.10583 \cdot 10^6$
5	0.17939	$2.80 \cdot 10^2$	$7.708 \cdot 10^3$

Tabela 11: Testes do parâmetro *tamanho do torneio*

O tamanho de torneio 3 permitiu uma melhora na *fitness* média pois proporcionou uma maior escolha do indivíduo vencedor do torneio na população a cada vez levando em conta que o tamanho máximo da população não é relativamente grande.

Por fim, foi comparado a diferença que o elitismo faz no algoritmo.

Elitismo	Melhor	Média	Pior
0	$3.456 \cdot 10^3$	$4.732 \cdot 10^3$	$9.034 \cdot 10^3$
1	0.16827	$1.664 \cdot 10^3$	$3.8989 \cdot 10^4$

Tabela 12: Testes do parâmetro *elitismo*

Sem o elitismo, o algoritmo não conseguiu manter e melhorar seus indivíduos, por isso caiu em uma busca aleatória, como comprovado pela uniformidade das *fitness* médias.

Com base nos testes feitos anteriormente, foi gerado um resumo dos parâmetros escolhidos, as *fitness* médias e o gráfico para os resultados obtidos quando executando o algoritmo com os dados do **Keijzer 10**:

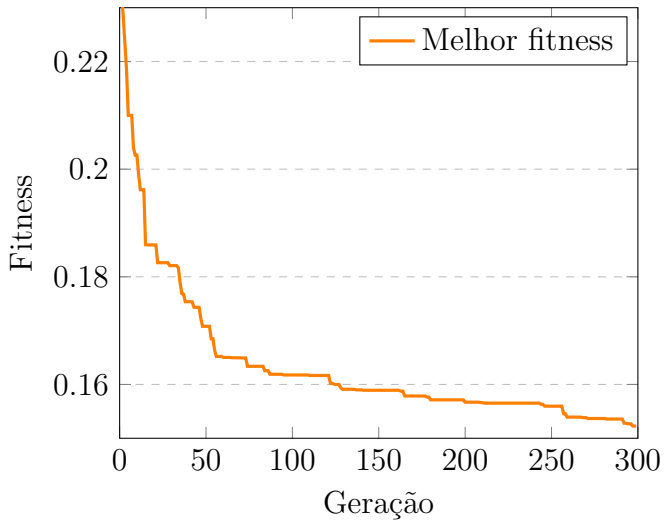
População	Geração	*Crossover*	Mutação	Torneio	Elitismo	População Inicial
200	300	0.9	0.1	3	1	50

Tabela 13: Resumo dos parâmetros utilizados

Melhor	Média	Pior
0.16368	$1.0332 \cdot 10^4$	$2.05538 \cdot 10^5$

Tabela 14: Resultado de *fitness* média para o *dataset* de teste

Gráfico *fitness* x geração Keijzer 10 Test



Este último gráfico, se comparado com o gráfico gerado na subseção anterior, mostra que para o **Keijzer 10** o algoritmo demorou mais para convergir e provavelmente buscou melhor uma solução no seu espaço de busca. Seu erro também se aproximou mais do esperado. Mesmo assim, os valores encontrados estão um pouco acima do esperado, significando que o algoritmo ainda não convergiu como deveria.

House

Por causa de um erro no algoritmo, os testes demoraram intervalos de tempo muito grandes para serem feitos. Em especial, um teste pequeno deste conjunto levou em torno de 5 horas para ser concluído. Então, os dados apresentados aqui são parciais para o que pode ser executado em tempo ábil.

Segue abaixo os testes base e as *fitness* médias encontradas para o conjunto de dados **House**:

População	Geração	Melhor	Média	Pior
10	20	$6.40499 \cdot 10^5$	$1.81297 \cdot 10^8$	$1.68749 \cdot 10^9$
10	50	$6.03851 \cdot 10^5$	$8.92938 \cdot 10^8$	$9.65257 \cdot 10^9$
10	100	$5.40322 \cdot 10^5$	$2.36913 \cdot 10^10$	$2.95810 \cdot 10^11$
20	20	$6.29823 \cdot 10^5$	$2.14218 \cdot 10^9$	$2.54557 \cdot 10^10$
20	50	$5.90125 \cdot 10^5$	$2.22771 \cdot 10^19$	$1.46654 \cdot 10^20$
20	100	$5.65421 \cdot 10^5$	$2.70514 \cdot 10^11$	$5.28916 \cdot 10^12$
30	20	$6.33832 \cdot 10^5$	$7.19283 \cdot 10^9$	$6.48892 \cdot 10^10$
30	50	$6.09453 \cdot 10^5$	$1.39465 \cdot 10^11$	$1.36566 \cdot 10^12$
30	100	$5.74728 \cdot 10^5$	$2.46394 \cdot 10^12$	$4.01612 \cdot 10^14$
15	300	$4.91181 \cdot 10^5$	$1.77081 \cdot 10^11$	$3.18721 \cdot 10^12$

Tabela 15: Testes dos parâmetros *população* e *geração*

Observando as médias das melhores *fitness*, pode-se notar que as soluções de erros menores vieram de algoritmos executados com maior número de gerações. Outros testes feitos e discussões com os colegas de classe mostraram que este *dataset* não precisa de uma população máxima grande desde que tenha gerações para ser capaz de evoluir e melhorar. Como uma confirmação deste fato, foi feito um teste extra com uma população de 15 indivíduos e 300 gerações.

Os testes quanto à diferença entre alta e baixa probabilidades de *crossover* também puderam ser executados e foram obtidos usando uma população de 10 indivíduos em 100 gerações:

Probabilidade de *crossover*	Melhor	Média	Pior
0.6	$5.80984 \cdot 10^5$	$4.65732 \cdot 10^9$	$1.80023 \cdot 10^10$
0.9	$6.27354 \cdot 10^5$	$3.40317 \cdot 10^11$	$1.20399 \cdot 10^12$

Tabela 16: Testes do parâmetro *probabilidade de crossover*

A diferença de valores para estes testes se dá porque a falta de diversidade devido ao tamanho da população é compensada com uma maior diversidade desta derivada da alta taxa de mutação.

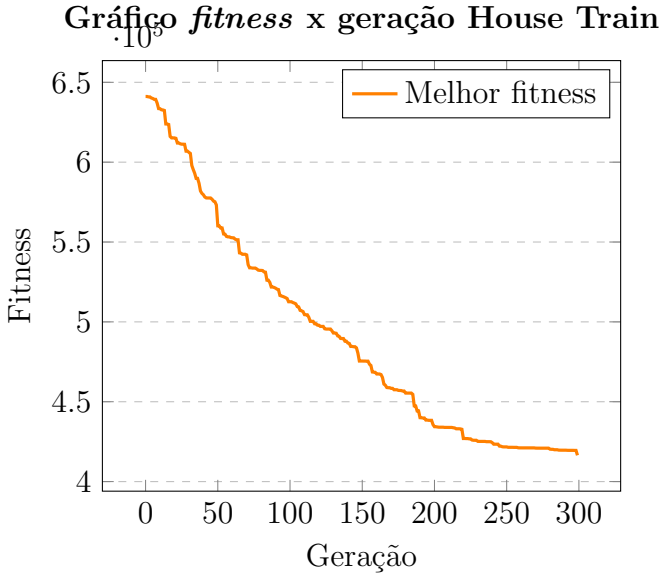
Para ilustrar a convergência do algoritmo para o *dataset* **House**, foi utilizado o teste base de melhor resultado. Um resumo dos parâmetros escolhidos, as *fitness* médias e o gráfico segue abaixo:

População	Geração	*Crossover*	Mutação	Torneio	Elitismo	População Inicial
15	300	0.95	0.05	2	1	5

Tabela 17: Resumo dos parâmetros utilizados

Melhor	Média	Pior
$4.91181 \cdot 10^5$	$1.77081 \cdot 10^1 1$	$3.18721 \cdot 10^1 2$

Tabela 18: Resultado de *fitness* média para o *dataset* de teste



Apesar da solução para este caso não ter sido ótima, ele ilustra muito bem a ação do número de gerações sobre a convergência do algoritmo. Observando o gráfico, é possível notar que o algoritmo utiliza bem o espaço de busca e que talvez com mais gerações fosse possível alcançar o ótimo mesmo com uma quantidade pequena de indivíduos.

Conclusão

O primeiro trabalho prático da disciplina de Computação Natural trata sobre a utilização da programação genética para a resolução de problemas de regressão simbólica de forma eficiente. Apesar do algoritmo deste trabalho não ter chegado a resultados ótimos, os conceitos vistos em sala de aula foram fortemente compreendidos e revisados.

Um problema que apareceu ao fim da implementação impediu que o programa chegasse a uma solução ótima para qualquer um dos *datasets*, apesar de chegar a uma solução aceitável para todos eles. Com o intuito de arrumar o defeito foi necessário passar por todos os aspectos que poderiam gerar o problema e testá-los um a um. No fim, ainda não foi totalmente entendido qual é a exata raiz do problema, apesar de que métodos como variação da pressão seletiva talvez sejam capazes de consertá-lo.

Bibliografia

- A Field Guide to Gennetic Programming
- 4. A Genetic Programming Introduction : Symbolic Regression
- Genetic Programming
- 9.4: Genetic Algorithm: Looking at Code - The Nature of Code