Algoritmos Genéticos

Vilmar Dorneles Aprato Neto & Theodoro Mota

Introdução:

O trabalho de implementação consiste em implementar duas funções matemáticas, utilizando algoritmos genéticos para conseguir otimizar os resultados. Para isso, foram escolhidas duas funções, sendo elas a função de Michalewicz [Figura 1.1] e a função *Six Hump* [Figura 1.2]

$$f(x) = -\sum_{i=1}^{n} \sin(x_i) \left[\sin(\frac{ix_i^2}{\pi})\right]^{2m}$$

[*1.1 - Michaleweicz's function]*

$$f(x_1, x_2) = (4 - 2.1x_1^2 + \frac{x_1^4}{3})x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2.$$

[1.2 - *Six Hump Camel*]

Ferramentas de desenvolvimento:

Para a implementação do projeto foi utilizada a linguagem *Python*, devido a sua facilidade de programação e legibilidade do código. Também foi utilizada o *Git*, para controle de versionamento, que auxilia na produção do código e compartilhamento de versões.

O nosso trabalho está disponível no endereço web: Algoritmo Genético

O algoritmo:

O algoritmo se baseia na estrutura de um algoritmo genético tradicional. Como o exemplo abaixo:

- ***Procedimento AG***
- ***{ t = 0;***
- ***inicia_população (P, t)***
- ***avaliação (P, t);***
- ***repita até (t = d)***
  - ***{ t = t +1;***
  - ***seleção_dos_pais (P,t);***
  - ***recombinação (P, t);***
  - ***mutação (P, t);***
  - ***avaliação (P, t);***
  - ***sobrevivem (P, t)***
  - ***}***
- ***}***

onde:
t - tempo atual;
d - tempo determinado para finalizar o algoritmo;
P - população

(Como visto em:http://conteudo.icmc.usp.br/pessoas/andre/research/genetic/)

O tempo pode ser expresso de uma forma diferente, como número de laços executados.

As etapas do algoritmos seguem a seguinte ordem:

Inicialização da população -> avaliação/*fitness* dos indíviduos -> seleção -> recombinação -> evolução -> mutação -> reprodução

No caso do nosso código utilizamos os seguintes parâmetros, já pré-definidos:

- INDIVIDUAL_LENGHT -> números de bits para representação de cada indivíduo
- POP_SIZE -> o tamanho da população
- SELECTED -> a porcentagem da população que será selecionada como pai para a próxima geração (apenas os melhores)
- RANDOM_IND -> a porcentagem de indivíduos aleatórios (para evitar máximos locais)
- MUTATE -> a porcentagem de indíviduos que sofrerão mutação
- MUTATE_FACTOR -> probabilidade de mudança de bit na mutação

O Nosso Algoritmo:

O nosso algoritmo utiliza como base o GA tradicional com algumas modificações que foram apropriadas para otimização de funções.

- Inicializamos uma população P.
- Evoluímos esta população
    - Calculamos o fitness de todos os indivíduos da população e ordenamos a lista de acordo com o fitness (o fitness é a função a ser otimizada).
    - Salvamos uma referência para o melhor indivíduo.
    - Para a nova população selecionamos 20% dos melhores indivíduos para ser pais.
    - Adicionamos mais 5% de indivíduos aleatórios como pais (para evitar máximos locais).
    - Mutamos 1% dos pais.
    - Fazemos crossover dos pais para completar os 75% restante da população com novos indivíduos.
    - Ao final temos a nova população.
- Continuamos evoluindo a população até que a referência ao melhor indivíduo não mude por 200 gerações (esse número poderia ser melhor, visto que na 4 geração ga encontramos o resultado final).

Inicialização da população:

A inicialização da população acontece ao criar números binários aleatórios em forma de string. A população inicial será a lista desses números aleatórios. O tamanho da população e o número de bits randômicos será determinados pelas constantes pré-definidas.

Evolução:

A evolução ocorre uma população diferente da recebida. Para isso, o primeiro passo é gerar uma lista de pares de cada indivíduo, composto pelo indivíduo e sua fitness. Ocorrerá um *sort* na lista, do pior até o melhor indivíduo. A partir dessa classificação, X% dos indivíduos serão os pais, depois disso serão criados indivíduos aleatórios para compor a nova população, além de incluir possíveis mutações. Com a lista dos pais calcula-se quantos indivíduos faltam para o próximo *pop*, através do seguinte cálculo.

```
number of children = (POP_SIZE - len(parents))
```

No loop será escolhido randomicamente um pai e uma mãe, se divide os bits na metade e se mistura essas duas metades para gerar um indivíduo. E logo em seguida, se entregará a

nova população.

Mutação:

Mutamos apenas 1% dos pais já selecionados antes do crossover, entretanto a mutação é bem agressiva visto que adicionamos o peso de 60% de mutação para cada bit do indivíduo a ser mutado. Ou seja, o indivíduo que entrar como parâmetro para a função mutante, terá 60% em cada bit de sofrer um *bitflip*, o que muda drasticamente o indivíduo. Como apenas os pais sofrem mutação, isso reflete drasticamente no crossover ajudando a escapar de máximos e mínimos locais.

Crossover:

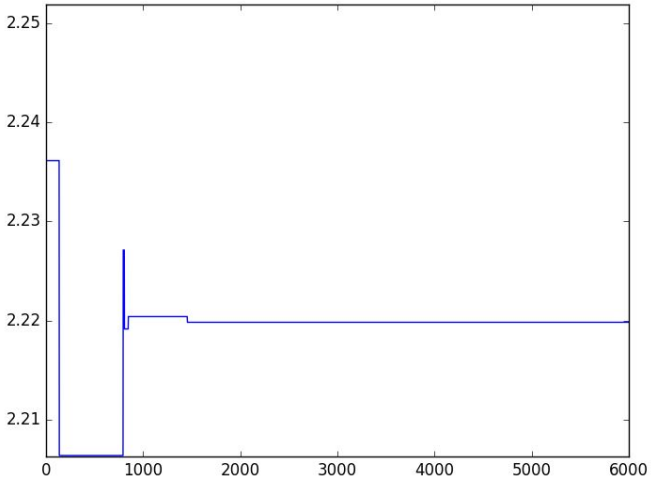
Selecionamos randomicamente 2 indivíduos da lista de pais, se eles não forem o mesmo pegamos metade do primeiro mais metade do segundo e geramos um novo indivíduo. Adicionamos todos os novos indivíduos na lista de pais até completar a nova população.

Resultados:

Para as duas funções escolhidas encontramos o mínimo local em no máximo quatro gerações. Isso se dá devido ao tamanho da população inicial e a maneira como fazemos o crossover e a seleção de pais. A execução dos programas demora menos de 2 segundos para um indivíduo representado com 15 bits para uma variável e 30 bits para duas.

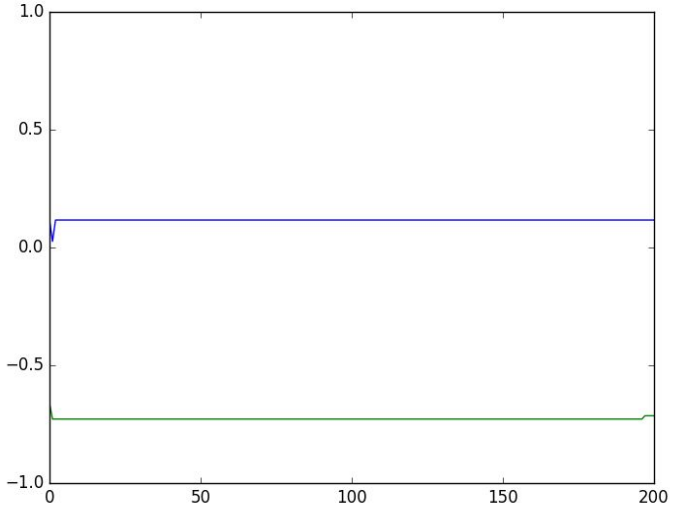
Nos gráficos a seguir temos o número de gerações em X e em Y o valor das variáveis que retornam o menor fitness.

Michaleweicz's function:



Valor x, aproximado em 2.1906.

Six Hump Camel:



Valor x1 = 0.11655, valor x2 = -0.679525

Conclusões:

Neste trabalho ficou evidente que GA é uma maneira rápida e fácil para otimizar funções, e devido a natureza das funções também foi possível perceber o quanto o tamanho inicial da população influencia na velocidade para encontrar a resposta. Outro ponto interessante de se analisar foi a maneira com que realizamos as mutações, isso aumentou muito a velocidade com que encontramos o resultado final.