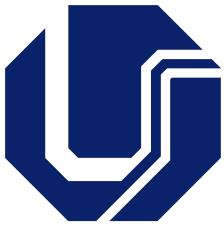
Algoritmos Genéticos

Antonio Fernandes Valadares

Trabalho 2



FEELT
Universidade Federal de Uberlândia
14 de dezembro de 2021

Introdução

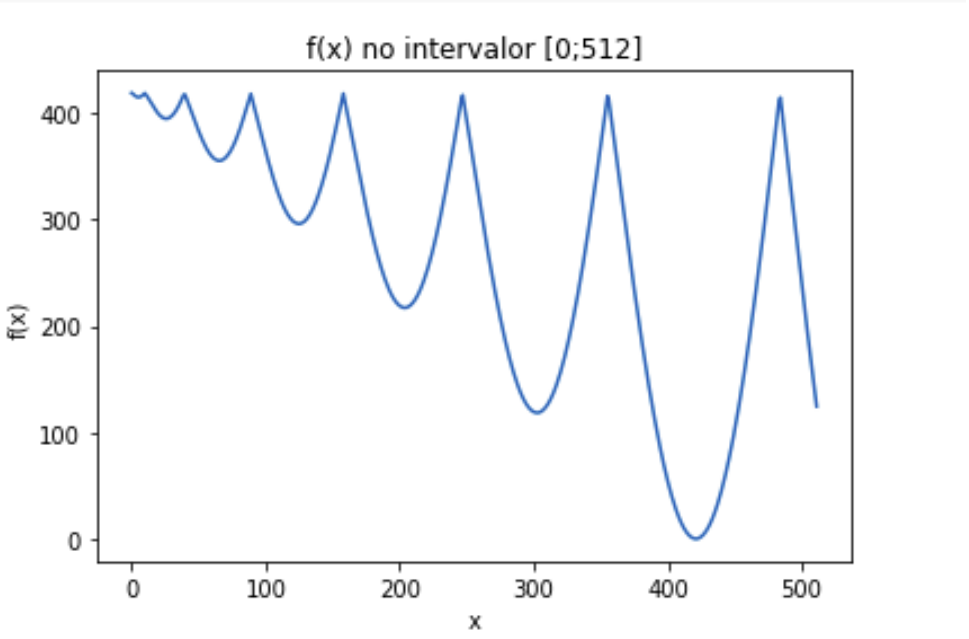
Esse trabalho foi desenvolvido seguindo a seção 3 do artigo *A Survey of Genetic Algorithms* do **M. Tomassini** a qual trata de um exemplo simples de utilização de algoritmos genéticos para otimização. Dessa forma foi densenvolvido um algoritmo para minimizar a seguinte função:

$$f(x) = -|x \sin(\sqrt{|x|})$$

O problema consiste em achar o x no intervalor [0;512] que minimize a função f. Para que no nosso algoritmo obtenhamos apenas valores positivos de fitness vamos somar uma costante, para que todos valores nesse intervalo sejam positivos, no caso foi somado uma constante de 419.

O código foi todo desenvolvido utilizando notebooks python.

$$f(x) = -|x \sin(\sqrt{|x|}) - 419$$



Construção do algoritmo

A população inicial será formada por 50 indivíduos aletoriamente escolhidos no intervalo de [0;512], porém como será necessário trabalhar como uma string binária para representar os valores de x. Então construi um vetor de 50 números inteiros de 0 a 1024 os quais

poderiam ser facilmente representados e numa string de 10 bits garantindo e para o cálculo da aptidão esses números seriam divididos por 2. Dessa forma o erro máximo que teremos é de 0.25. Para a construção desse vetor utilizei a biblioteca numpy.

```
1  initial_population =
2    np.random.choice(np.arange(0,1024), replace=False, size=(50))
```

Após a criação da população inicial, foi definida a função objetivo:

```
1  def f(x):
2    return  (-1 * np.abs(x * math.sin(math.sqrt(np.abs(x)) ) )) + 419
```

Defini uma função para o cálculo do fitness de toda população, também com a ajuda do numpy que mapeia uma função para todos os valores do array, importante dividir os valores por 2.

```
1  def fitness_calculate(population):
2
3    population = population/2
4
5    fitness = np.vectorize(f)(population)
6
7    return fitness
```

Então, com os resultados o vetor é ordenado de forma com que o melhor resultado esteja primeiro, além disso o melhor indivíduo e seu valor é exibido.

```
1  def best_result(population, fitness):
2    result = np.column_stack((population, fitness))
3    result = result[result[:, 1].argsort()]
4    best = result[0]
5
6    return result
```

Os indivíduos selecionados para realizar o crossover, foram uma porcentagem dos melhores indíviduos. Não é uma forma ideal de fazer a seleção pois é determinística e o algoritmo pode se perder em um mínimo local. Porém ainda pretendo implementar um metódo de seleção de roleta ou algum outro metódo.

```
1  def select_individuals(result):
2    select_ind = result[0:20, 0] * 2
3    bin_population = ["{0:010b}".format(int(x)) for x in select_ind]
4
```

```
5     return bin_population
```

Para a geração da nova população foi feito um crossover entre os indivíduos selecionados e então todos foram aplicados a uma probabilidade de mutação. Foram realizados vários testes para a geração da nova população algumas vezes os indivíduos selecionados eram incluidos na nova geração. Foi executado testes incluindo 5 ou 10 novos indivíduos aletórios na nova geração, também foram realizados testes apenas com indivíduos gerados pelo crossover.
O crossover foi realizado escolhendo dois indivíduos selecionados aleatoriamente e escolhendo um número também aleatório entre 0 e 10 para cortar as strings de bits, para cada par de pais são gerados um par de filhos.

```
1   def crossover(population):
2     new_population = []
3
4     for element in population:
5       new_population.append(element)
6
7     while(len(new_population) < 45):
8       p1 = random.choice(population)
9       p2 = random.choice(population)
10
11      i = random.randint(0,10)
12
13      f1 = p1[0:i] + p2[i:10]
14      f2 = p2[0:i] + p1[i:10]
15
16      new_population.append(f1)
17      new_population.append(f2)
18
19    for i in range(0, 5):
20      new_population.append("{0:010b}".format(int(random.randint(0,1024)))
21
22    return new_population
```

Então para cada novo indivíduo foi aplicado uma possibilidade mutação que alteraria um bit aleatório da string. Foram realizados teste com 10%, 5%, 1%. No geral valores não tão altos tiveram resultados mais satisfatórios. Foram realizados testes sem a mutação também e o desempenho caiu consideravelmente.

```
1   def mutation(population):
```

```
2       new_population = []
3
4       for element in population:
5         if(random.random() < 0.05):
6           new_element = list(element)
7           pos = random.randint(0,9)
8           if  (new_element[pos] == '1'):
9             new_element[pos] = '0'
10          else:
11            new_element[pos] = '1'
12          new_population.append(''.join(new_element))
13        else:
14          new_population.append(element)
15
16      return new_population
```

Vários quantidade de gerações foram testadas, para números muito grandes sempre era obtido o resultado ótimo. No geral, com 10 gerações o algoritmo encontra a maioria das vezes o valor de 421, que é o mínimo local.

Conclusão

Apesar da seleção de indivíduos não ter sido feita da forma adequada, o algoritmo teve bons resultados no geral. Acrescentar novo indíviduos aleatórios na geração seguinte fez com que o algoritmo evitasse de cair em mínimos locais apesar de que a soma da aptidão dos indivíduos não convergia. Foram feitos testes escolhendo os 10, 15, 20, 25 melhores indivíduos, aparentemente os melhores resultados foram com 20.

Foi muito interessante criar uma solução utilizando um algoritmo genético, pois é possível observar a evolução do resultado com o passar das gerações e são várias váriaveis e parâmetros para serem testados. Além de existirem várias possibilidades diferentes de se realizar os processos de seleção, crossover e mutação.