NOME: VANDO CARLOS DINIZ REIS - 2019057195
NOME: RAFAEL MOURA MARQUES - 2019089305
TÍTULO: Weasel Program

INTRODUÇÃO:

O 'Weasel Program' é uma experiência feita para demonstrar como uma população de strings pode evoluir até convergir para uma string específica, no caso:

'ME*THINKS*IT*IS*LIKE*A*WEASEL'

Esse programa é inspirado no 'teorema do macaco infinito', que diz que um macaco digitando aleatoriamente em um teclado por um intervalo de tempo infinito, eventualmente irá criar qualquer texto desejado, como por exemplo uma obra de William Shakespeare. Entretanto, esse cenário é praticamente impossível, pois a chance do macaco digitar a frase de 29 caracteres que queremos é praticamente nula.

Podemos contornar esse imprevisto com um algoritmo genético computacional, pois além de digitar inúmeros indivíduos de 29 caracteres em uma velocidade incrivelmente mais rápida, os descendentes estão sujeitos a mutações aleatórias, que diversificam a população, e consequentemente agilizam ainda mais o nosso objetivo. Uma ou mais mutações podem ser as responsáveis pelo aparecimento dos nossos caracteres desejados em uma troca de gerações. [Resposta da pergunta número 2]

O programa gera aleatoriamente uma string de 29 caracteres e realiza N cópias da mesma. Entretanto, cada cópia possui uma certa chance de mutação em cada um dos caracteres. Feito isso, ele avalia todos os indivíduos da população e seleciona o mais próximo da frase desejada. Ele será a string base para a próxima geração e o processo se repete até que a frase seja alcançada.

Exemplo:

Geração 1: WDLTMNLT DTJBKWIRZREZLMQCO P
Geração 2: WDLTMNLT DTJBSWIRZREZLMQCO L
Geração 10: MDLDMNLS ITJISWHRZREZ MECS L
Geração 20: MELDINLS IT ISWPRKE Z WECSEL
Geração 30: METHINGS IT ISWLIKE B WECSEL
Geração X: METHINKS IT IS LIKE A WEASEL

IMPLEMENTAÇÃO:

- Representação escolhida:

  Um indivíduo é representado por uma string de 29 caracteres, podendo ser dígitos, letras maiúsculas ou espaços. Os espaços são representados pelo símbolo '*'.

  Exemplo: " 9AUMH8NXDJVT39V*Z5LC*2*WE5QEL

  Já a população é representada por uma lista de 100 indivíduos, chamada 'pop'. Para mais detalhes olhar as funções `def criaString():` e `def criaPop(indivíduo):` no código disponibilizado.

  Nota: *Como temos 37 opções disponíveis para um caractere, a chance de inicializarmos com um indivíduo perfeito é de 1 em 37^29. Ou seja, praticamente nulas!* *[Resposta da pergunta número 1]*

- Função de aptidão:

  A função de aptidão define a qualidade de uma solução candidata. Para o tipo de representação adotado, bastou-se comparar os caracteres da string selecionada com os caracteres da string desejada. Cada caractere semelhante garante aquele indivíduo 1 ponto.
  Já para analisar a aptidão da população, é preciso usar essa função em todos os elementos de 'pop'. Esses dados são armazenados em uma outra lista chamada de 'pop_fitness'.
  Para mais detalhes olhar as funções `def analisaCaso(palavra, gabarito=GABARITO):` e `def popFitness(pop):` no código disponibilizado.

- Operadores de Variação:

  Crossover: Cria 100 cópias idênticas da solução candidata, que substituirão a geração anterior.
  Mutação: Preserva o primeiro indivíduo da população e atua nos demais. Cada caractere dos 99 indivíduos restantes possui 5% de chance de sofrer uma mutação. Caso ele sofra, o caractere é substituído, obrigatoriamente, por um outro. Como strings são imutáveis em Python, foi necessário transferir todos os caracteres de um indivíduo para uma lista auxiliar, modificar os caracteres necessários, e criar uma nova string a partir dessa lista.
  Para mais detalhes olhar as funções `def criaPop(indivíduo):` e `def mutation(pop, MUTATION_RATE):` no código disponibilizado.

- ## Operadores de Seleção:

  Como não foi imposto nenhum método específico no roteiro, optamos pelo uso da seleção elitista. Ela consiste em selecionar o melhor indivíduo da geração para reintroduzi-lo na geração seguinte, dessa forma evitando perdas de informações importantes que poderiam ser perdidas durante o processo de evolução. Portanto, pegamos o indivíduo com o melhor fitness da população para ser o progenitor da próxima geração. Fizemos isso com o método `max(pop_fitness)` da linguagem Python.

- ## Gráfico:

  Ao final do programa, é gerado um gráfico informativo do melhor fitness ao longo das gerações. Para que ele funcione é necessário instalar e importar a biblioteca `matpltlib.pyplot`. Qualquer erro na importação dessa biblioteca não compromete o funcionamento do código, apenas não será mostrado o gráfico ao final do programa. O programa coleta informações do melhor indivíduo e das gerações enquanto é executado e as armazena em duas listas separadas. Essas listas são usadas para plotar o gráfico. Para mais detalhes olhar a função `def grafico(geracoes, bests):` no código disponibilizado.

- ## Condição de Parada:

  O código termina somente quando a solução ótima é encontrada.

- ## Código Principal:

```python
#ALGORITMO GENÉTICO COMPLETO
bests = list()                                          #cria uma lista para armazenar dados
geracoes = list()                                       #cria uma lista para armazenar dados
palavra = criaString()                                  #gera uma palavra aleatória
best = analisaCaso(palavra)                             #analisa o fitness da palavra
pop = [palavra]                                         #população inicial contém apenas a palavra sorteada
i = 0                                                   #contador de gerações
print(f'String inicial: {palavra}')
while best!=TAMANHO:
    i+=1                                                #incrementa um ao número da geração
    if(rd.random()<CROSSOVER_RATE):                     #analisa se tem cruzamento ou não
        pop = criaPop(palavra)                          #cria as 100 cópias
        pop = mutation(pop, MUTATION_RATE)              #aplica as mutações nas 100 cópias
    pop_fitness = popFitness(pop)                       #analisa o fitness da população
    best = max(pop_fitness)                             #descobre o melhor fitness
    pos_best = pop_fitness.index(best)                  #descobre a posição do melhor fitness
    palavra = pop[pos_best]                             #descobre o indivíduo com melhor fitness
    bests.append(best)                                  #adiciona informação na lista de dados
    geracoes.append(i)                                  #adiciona informação na lista de dados
    if(i%10 == 0):
        print(f'Melhor indivíduo da {i}ª geração: {palavra}')       #printa o melhor indivíduo a cada 10 gerações
#INFORMAÇÕES FINAIS
print(f'String final: {palavra}')
print(f'Número de caracteres iguais: {best}')
print(f'Geração final: {i}ª')
try:
    grafico(geracoes,bests)
except:
    print('Erro ao gerar gráfico!')
```

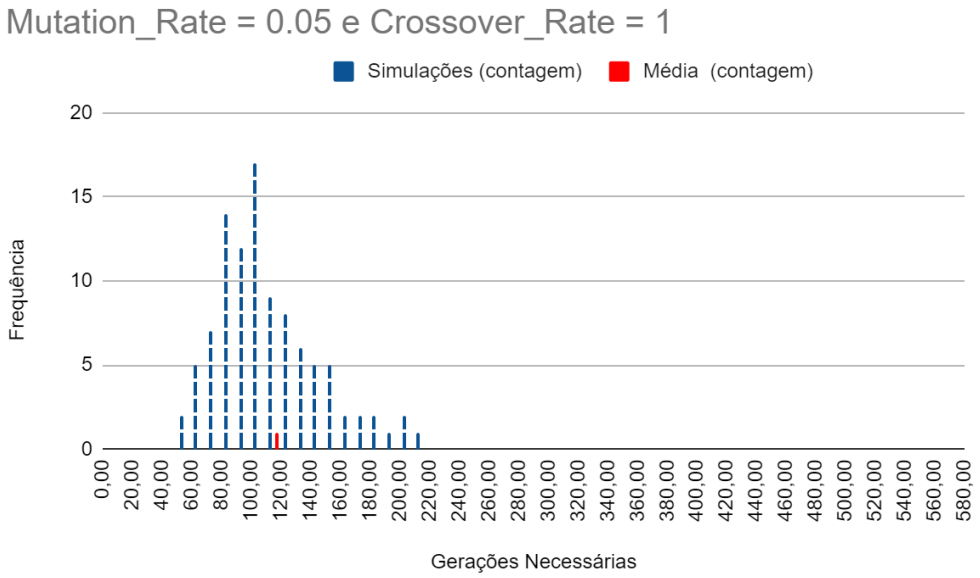
- ## Testes:

Todas as funções criadas foram testadas no bloco seguinte para garantir o sucesso do código.

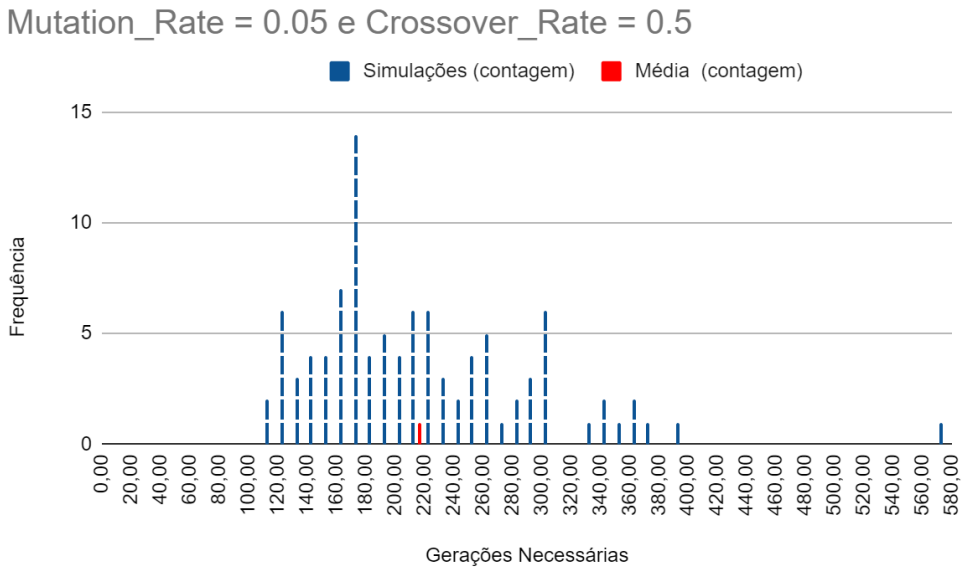
ANÁLISE DOS RESULTADOS:

- Crossover Rate:

Para testarmos a implicancia de mudanças na taxa do `crossover_rate` no número de gerações necessárias para chegar ao gabarito decidimos simular o código 100 vezes para duas taxas de crossover diferentes (1 e 0.5). Obtendo assim os gráficos abaixo:

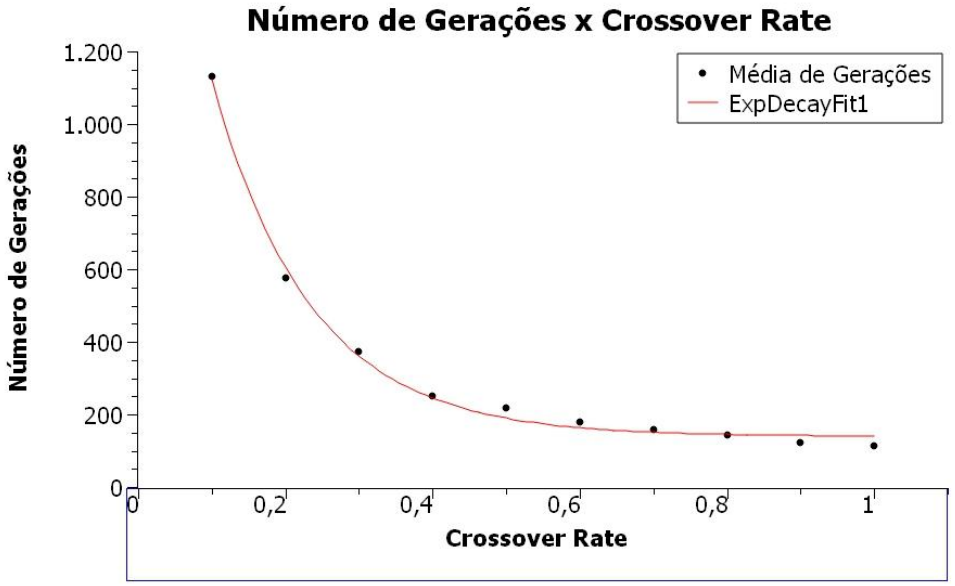


Podemos analisar então que no primeiro gráfico (`crossover_rate` = 1), no qual acontece o crossover em todas as gerações, a média de gerações necessárias para alcançarmos a solução correta foi 112,76.



Já no segundo gráfico (crossover_rate = 0,5), o crossover só acontece em aproximadamente metade dos casos. Isso faz com que o gráfico, de modo geral, seja deslocado para a direita e que a média das gerações com solução aumente drasticamente, sendo igual a 218,74.

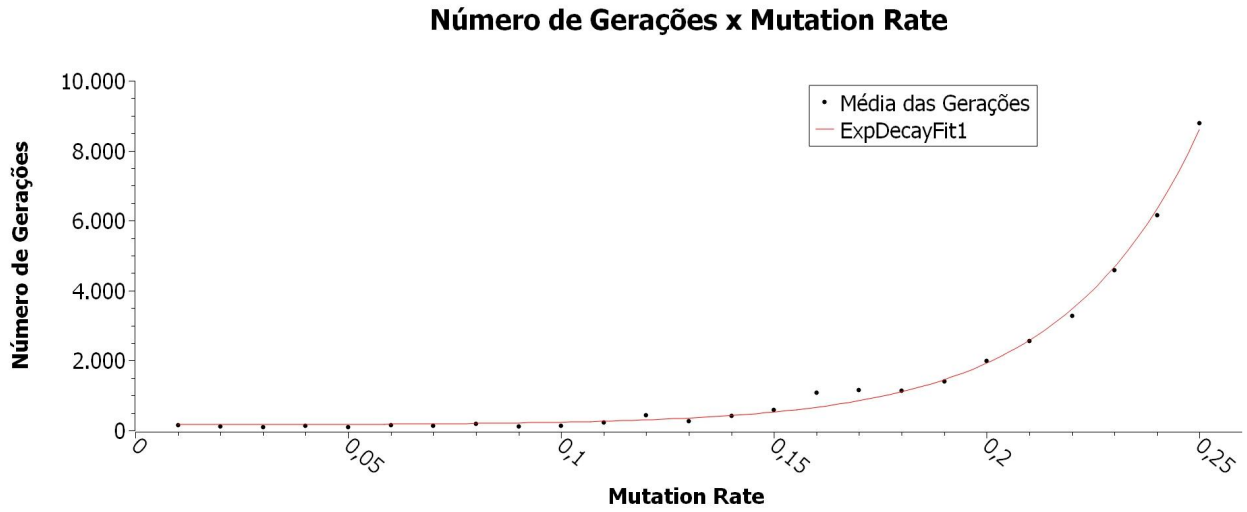
Conclui-se então que quanto mais alto o crossover rate, mais rápido e de maneira exponencial a população converge para a solução desejada. Simulando 100 vezes o programa, para cada Crossover Rate em um intervalo de 0.1, chegamos ao seguinte gráfico:



- Mutation Rate:

Com o objetivo de observarmos como a taxa de mutação implica nas simulações optamos por simular 100 vezes para valores de mutation_rate entre 0,01 e 0,25.

Obtendo, portanto, a média de gerações necessárias em cada uma das taxas para convergir para a string desejada.
No gráfico abaixo iremos demonstrar uma curva que contém a média de gerações para cada taxa:



Conclui-se que quanto maior a taxa de mutação, mais gerações são necessárias para a convergência. Pelo gráfico, é possível perceber que isso ocorre de forma exponencial. Isso se deve, pois uma alta taxa de mutação implica numa frequente troca de caracteres, inclusive os corretos. Assim, as características herdadas das gerações são facilmente perdidas, ficando aquém da sorte para alcançar a string desejada. E como vimos anteriormente, a chance de gerar a string correta aleatoriamente é extremamente baixa.

Considerações finais:

- Após toda essa pesquisa, percebe-se que o algoritmo mais eficiente é aquele com `crossover_rate` = 1 e `mutation_rate` = 0.05. A média de gerações para a convergência nesse caso é de 112,76 gerações!
- As funções têm seu funcionamento e lógica explicadas no arquivo .pynb.