Jaler Samuel Zago - 108352

Sistemas Inteligentes

setembro de 2019

Adaptação de código em python para uso de torneio

INTRODUÇÃO

Nas últimas aulas de Sistemas Inteligentes tem-se mostrado algoritmos genéticos e a suas aplicações na área da computação. O score é uma métrica, que leva em consideração várias características e é usado para medir os critérios de modo a escolher sempre os indivíduos que melhor se adaptam ao meio. Dentre algumas das técnicas possíveis para fazer uma escolha mais eficaz e obter os indivíduos mais adaptados, ou seja, seleção natural, encontra-se o método do torneio, que será utilizado neste trabalho para substituir a aleatoriedade já implementada no algoritmo original.

DESCRIÇÃO

Foi apresentado para a turma, um algoritmo genético programado em python que dá uma ideia ampla de como funciona a seleção natural. O objetivo desse algoritmo é, determinar a menor média fitness, ou seja, dado um conjunto de genes (inteiros) somados - “sum” e um “target”. Fazer uma média do resultado de toda a população.

Inicialmente o algoritmo dá valores as variáveis e cria uma população com o número de genes pré definido e selecionados de forma aleatória. A quantidade de indivíduos na população também é pré definida. Em seguida, ele calcula a primeira média fitness e adiciona a “fitness_history”.

```
target = 352
i_length = 10
i_min = 0
i_max = 1000
p_count = 100
epochs = 100
p = population(p_count, i_length, i_min, i_max)
fitness_history = [media_fitness(p, target),]
```

Figura 1: Iniciação dos valores das variáveis, determinação da população inicial e cálculo da primeira média geral de fitness.

```
def individual(length, min, max):
    'Create a member of the population.'
    return [ randint(min,max) for x in range(length) ]

def population(count, length, min, max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values

    """
    return [ individual(length, min, max) for x in range(count) ]
```

Figura 2: Funções usadas para determinar uma população com genes aleatórios.

```
def fitness(individual, target):
    """
    Determine the fitness of an individual. Higher is better.

    individual: the individual to evaluate
    target: the target number individuals are aiming for

    O fitness do individuo perfeito sera ZERO, ja que o somatorio dara o target
    reduce: reduz um vetor a um escalar, neste caso usando o operador add
    """
    sum = reduce(add, individual, 0)
    return abs(target-sum)

def media_fitness(pop, target):
    'Find average fitness for a population.'
    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)
```

Figura 3: Funções usadas respectivamente para determinar a fitness de um indivíduo e para determinar a média fitness da população inteira.

Depois disso, é chamado um laço de iteração para a quantidade de épocas que foi pré definido o algoritmo. Nesse laço, a nova população será sempre o “return” da função “evolve”. A cada nova população, uma nova média fitness é criada e adicionada a “fitness_history”.

```
for i in range(epochs):
    p = evolve(p, target)
    fitness_history.append(media_fitness(p, target))
```

Figura 4: Laço de iteração determinado pelo número de épocas.

Na função “evolve”, o fitness de cada indivíduo é calculado e guardado na lista “graded”, então é usado a função interna do python “sorted” para ordenar eles, do menor para o maior. Baseado na porcentagem escolhida para a variável retain, o top da lista “graded” é adicionado à lista de pais - Elitismo.

```
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    'Tabula cada individuo e o seu fitness'
    graded = [ (fitness(x, target), x) for x in pop]
    'Ordena pelo fitness os individuos - menor->maior'
    graded = [ x[1] for x in sorted(graded)]
    'calcula qtos serao elite'
    retain_length = int(len(graded)*retain)
    'elites ja viram pais'
    parents = graded[:retain_length]
```

Figura 5: Função “evolve”, representando o elitismo.

Além da elite selecionada por classificação de score, no algoritmo apresentado em aula, uma parte do restante da população é aleatoriamente, utilizando a probabilidade de acontecer (random_select), escolhida para fazer parte dos pais também.

```
for individual in graded[retain_length:]:
'garante que a população restante tenha n% de chance de virar pais, mesmo nao sendo elite'
    if random_select > random():
        parents.append(individual)
```

Figura 6: Iteração que garante à população, não selecionados no elitismo, ter random_select % de chance de ser colocada na lista de pais.

Após selecionados os pais para a próxima geração, aplica-se uma mutação em apenas 1 gene do indivíduo, determinada se vai ou não acontecer pela variável “mutate”.

```
for individual in parents:
    if mutate > random():
        pos_to_mutate = randint(0, len(individual)-1)
        # this mutation is not ideal, because it
        # restricts the range of possible values,
        # but the function is unaware of the min/max
        # values used to create the individuals,
        individual[pos_to_mutate] = randint(min(individual), max(individual))
```

Figura 7: Iteração que realiza a mutação nos genes dos pais.

Após a mutação, determina-se quantos pais já foram adicionados a lista e quantos filhos terão que ser adicionados para completar o número total da população.

Com o número de filhos que precisam ser gerados calculado, é selecionado aleatoriamente duas posições na lista de pais, uma para ser pai e outra para ser mãe. Garante-se que o número sorteado não seja o mesmo, então gera-se um filho com a metade inicial de genes do pai, e a metade final de genes da mãe. Depois, adiciona-se o filho gerado, a lista de filhos. Esse processo repete-se até que o número de indivíduos pertencentes a população seja satisfeita.

```
parents_length = len(parents)
'descobre quantos filhos terao que ser gerados alem da elite e aleatorios'
desired_length = len(pop) - parents_length
children = []
'comeca a gerar filhos que faltam'
while len(children) < desired_length:
    'escolhe pai e mae no conjunto de pais'
    male = randint(0, parents_length-1)
    female = randint(0, parents_length-1)
    if male != female:
        male = parents[male]
        female = parents[female]
        half = len(male) // 2
        'gera filho metade de cada'
        child = male[:half] + female[half:]
        'adiciona novo filho a lista de filhos'
        children.append(child)
```

Figura 8: Representação do nascimento de um filho com genes do pai e da mãe, respectivamente .

Enfim, a lista de filhos é incorporada a lista de pais e retornada ao usuário como uma nova população. Nesse processo os pais selecionados na figuras 5 e 6, foram clonados para a próxima geração.

```
'adiciona a lista de pais (elites) os filhos gerados'
parents.extend(children)
return parents
```

Figura 9: Clonagem dos pais e retorno para uma nova população.

Após reproduzir todas as gerações, e adicionar todas as média fitness à lista “fitness_history” como representado na figura 4, o algoritmo mostra ao usuário as médias fitness de todas as gerações calculadas.

```
for datum in fitness_history:
    print (datum)
```

Figura 10: Mostra das médias fitness adquiridas em todas as gerações.

PROBLEMA

O problema consiste em retirar o método de seleção totalmente aleatório e implementar o método de torneio.

O torneio é um método consideravelmente eficaz para selecionar indivíduos de melhor score. De maneira geral, ele seleciona aleatoriamente n indivíduos e os compara, o que tiver o melhor score será utilizado. O n é pré definido.

RESOLUÇÃO

Para implementação do método proposto, inicialmente desenvolvi uma função “torneio” onde os parâmetros de entrada são: a lista na qual os indivíduos devem ser sorteados, a posição a partir de

onde, na lista, deve ser iniciado o sorteio, a posição até onde o sorteio deve ir, o “target” como métrica de avaliação, e o número de indivíduos sorteados por torneio.

O algoritmo consiste em sortear um indivíduo e colocá-lo na lista “listaSorteio”. Então, pega-se a fitness dele e compara-se com a do indivíduo de melhor fitness. Se esse for o melhor (menor, neste caso), o melhor valor de fitness é atualizado e o “n” atual é guardado. Essa sorteio é repetido “n” vezes.

Após realizar “n” iterações, teremos o melhor “n” guardado, então basta pegar a “listaSorteio” na posição do melhor “n” guardado e teremos a posição do indivíduo que ganhou o sorteio, ou seja, o vencedor do torneio. Então basta-se retornar essa posição.

```
def torneio(lista,inicioLista,fimLista,target,n):
    'n é o número de sorteados para o torneio'
    listaSorteio=[0]*n
    cont=0
    menorFitness=999999999
    while cont<n:
        'sorteia um numero da lista e coloca na listaSorteio'
        listaSorteio[cont] = randint(inicioLista,fimLista)
        'pega a fitness desse individuo e coloca na lista para fazer o torneio'
        fitnessAtual = fitness(lista[listaSorteio[cont]],target)
        'pega o menor fitness entre os escolhidos pro torneio'
        if (fitnessAtual < menorFitness):
            menorFitness=fitnessAtual
            'guarda a posição do melhor score'
            melhorPosicao=cont
        cont+=1
    'retorna a posicao da lista dentre os n escolhidos, de melhor score'
    return listaSorteio[melhorPosicao]
```

Figura 11: Função “torneio”, explicada previamente.

Na função “evolve” utilizei três vezes a chamada da função torneio para substituir a escolha aleatória.

A primeira chamada acontece após selecionar a elite para determinar quais indivíduos, não contidos na elite, também devem ser pais. É ocupado a mesma iteração do algoritmo original, como mostrado na figura 6, para garantir que todo o resto da população tenha chance de ser pai, porém antes de colocar aleatoriamente um indivíduo na lista de pais é aplicado o método torneio para que, dentre "n", só o melhor seja selecionado.

```
for individual in graded[retain_length:]:
'garante que a população restante tenha n% de chance de virar pais, mesmo nao sendo elite'
    if random_select > random():
        'seleciona a posição do melhor individuo'
        melhorPosicao=torneio(graded,retain_length,len(graded)-1,target,n)
        'pega os genes do melhor individuo'
        individualMenor = graded[melhorPosicao]
        'adiciona ele a lista'
        parents.append(individualMenor)
```

Figura 12: Modificação feita para garantir que a população restante seja elegida por meio de torneio.

As outras duas chamadas da função ocorrem ao selecionar um pai e uma mãe na lista de pais, que era anteriormente aleatório. Agora faz-se um torneio, seleciona-se a melhor posição dentre os sorteados (tanto para o pai, quanto para a mãe) e verifica se o torneio não elegeu o mesmo indivíduo. Se a escolha for diferente, então é buscado os genes de ambos e cruzado, de forma a ser a primeira metade do pai e a segunda metade da mãe.

```
while len(children) < desired_length:
    'escolhe a melhor posicao para pai e mae por torneio'
    melhorPosicaoMale = torneio(parents, 0, parents_length-1, target, n)
    melhorPosicaoFemale = torneio(parents, 0, parents_length-1, target, n)
    if melhorPosicaoMale != melhorPosicaoFemale:
        male = parents[melhorPosicaoMale]
        female = parents[melhorPosicaoFemale]
        'calcula metade dos genes'
        half = len(male) // 2
        'gera filho metade de cada'
        child = male[:half] + female[half:]
        'adiciona novo filho a lista de filhos'
        children.append(child)
```

Figura 13: Modificação feita para selecionar, por meio de torneio, um pai e uma mãe.

CONCLUSÃO

A fim de testar o algoritmo e como solicitado no exercício,usei os três últimos dígitos da minha matrícula (352) como “target”, fiz um teste e apresento as saídas a seguir:

Na primeira execução, usando n=3 para fazer o torneio e o número de épocas igual a 100 obtive valores que variaram de 4706,58 na primeira época, até 617,00 na centésima época.

```
C:\Users\Jaler\Desktop\Trabalho sistemas inteligentes>python teste.py
[4706.58, 3438.78, 1992.92, 1228.09, 1199.0, 1190.6, 1155.19, 1140.22, 1139.0, 1139.0, 1140.28, 1
139.0, 1141.29, 1139.0, 1142.93, 1141.16, 1139.0, 1139.0, 1139.0, 1139.0, 1139.0, 1139.0, 1139.0,
 1139.0, 1139.0, 1139.0, 1140.98, 1139.0, 1141.4, 1139.0, 1139.0, 1139.0, 1139.06, 1139.0, 1122.9
, 1001.17, 913.6, 909.0, 909.0, 909.0, 909.0, 909.37, 911.35, 909.0, 909.0, 909.0, 909.0, 909.11,
 912.43, 911.11, 909.0, 909.0, 911.42, 909.0, 909.0, 909.0, 909.0, 894.28, 807.56, 787.0, 787.0,
787.0, 787.0, 787.0, 787.0, 787.0, 787.0, 787.0, 787.0, 787.0, 787.0, 778.53, 730.04, 670.69, 666
.0, 666.0, 668.3, 660.24, 625.68, 618.0, 618.0, 618.0, 618.0, 618.0, 618.0, 618.0, 618.0, 618.0,
619.4, 620.29, 618.0, 617.89, 617.28, 617.0, 617.0, 617.0, 617.0, 617.0, 617.0, 617.64, 617.0]
```

Figura 14: Primeira execução do algoritmo modificado.

Na segunda execução, utilizei o mesmo n, porém mudei para 300 épocas. Percebi grandes melhorias, de uma primeira época com 4595,73, ele foi para 2,21 na trecentésima época.

```
C:\Users\Jaler\Desktop\Trabalho sistemas inteligentes>python teste.py
[4595.73, 3018.19, 2143.08, 1733.81, 1667.31, 1668.43, 1665.0, 1665.0, 1665.0, 1665.0, 1665.0, 16
66.55, 1665.0, 1665.0, 1653.75, 1600.5, 1590.0, 1590.0, 1594.26, 1593.0, 1590.0, 1590.0, 1595.75,
 1590.0, 1593.96, 1590.0, 1590.62, 1590.0, 1590.0, 1590.0, 1581.03, 1515.75, 1491.0, 1491.0, 1495
.39, 1440.27, 1165.29, 1068.32, 1068.0, 1068.0, 1069.5, 1068.0, 1068.0, 1068.0, 1068.0, 1068.0, 1
068.0, 1068.0, 1068.0, 1068.0, 1068.0, 1069.44, 1068.0, 1068.0, 1068.0, 1068.0, 1068.0, 1068.0, 1060.85,
1024.0, 1013.0, 1013.0, 1013.0, 1014.19, 1011.35, 998.81, 980.33, 980.0, 980.0, 980.0, 966.3, 882
.73, 838.45, 801.4, 778.0, 750.57, 607.19, 567.77, 567.0, 561.76, 506.74, 442.55, 436.43, 436.0,
436.0, 436.0, 436.42, 436.0, 436.0, 436.0, 436.0, 436.0, 436.0, 436.0, 436.21, 436.0, 436.0, 436.
72, 436.0, 437.37, 436.0, 436.0, 437.33, 427.38, 372.86, 359.0, 359.0, 359.0, 359.0, 359.0, 359.0
, 359.0, 359.0, 359.0, 359.0, 359.0, 360.81, 359.0, 359.0, 359.64, 359.58, 359.0, 359.0, 359.0, 3
59.0, 359.0, 359.0, 359.83, 359.0, 359.0, 359.0, 359.0, 359.0, 359.0, 359.0, 359.7, 359.85, 360.7
, 359.28, 359.0, 354.1, 328.62, 310.0, 310.0, 310.0, 310.0, 310.0, 310.0, 310.0, 310.0, 310.0, 31
0.0, 310.0, 308.32, 300.67, 296.0, 296.0, 297.07, 296.0, 296.0, 296.0, 294.88, 288.02, 282.14, 28
2.0, 282.0, 283.53, 278.24, 259.43, 235.0, 235.0, 235.0, 235.0, 235.56, 235.0, 235.0, 235.0, 235.
0, 236.59, 236.92, 235.0, 235.0, 235.0, 235.0, 235.0, 236.56, 235.0, 235.0, 235.0, 232.69, 216.94
, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.
0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 215.65, 214.0, 214.0, 214.02, 214.0, 215.32, 214.0, 214.0,
214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 214.0, 211.66, 199.78, 197.47, 196.0, 196.0, 196.0, 196.0, 196
.0, 196.0, 196.0, 196.0, 196.0, 196.0, 196.0, 196.0, 196.86, 196.0, 196.0, 196.0, 196.0, 197.51,
196.0, 196.0, 197.29, 196.0, 196.0, 196.0, 196.0, 196.0, 196.0, 196.0, 197.13, 196.0, 196.62, 196
.0, 196.95, 196.0, 196.0, 169.96, 43.48, 10.0, 10.0, 9.44, 5.72, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.
0, 2.0, 2.0, 2.23, 2.0, 2.0, 2.04, 2.0, 2.0, 2.31, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2
.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.21]
```

Figura 15: Segunda execução do algoritmo modificado.

Por último, mantive as 300 épocas e alterei o n para 8. O resultado final não expressou ganho significativo, mas os resultados intermediários melhoraram mais rápido. Isso é explicado por que quanto maior o número de participantes de um torneio, maior a chance de um indivíduo com score alto aparecer.

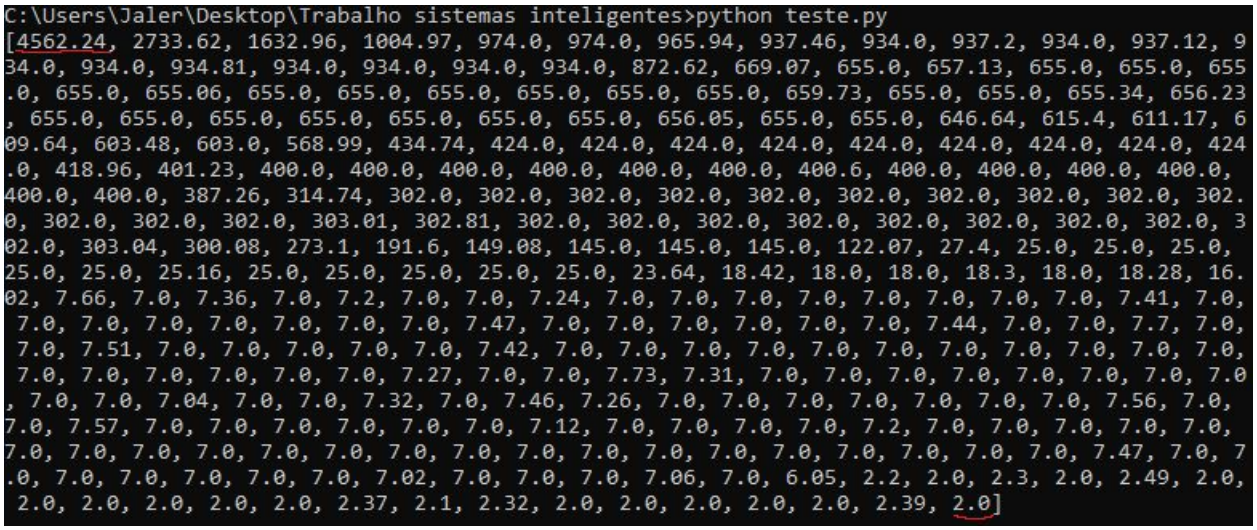

Figura 16: Terceira execução do algoritmo modificado.