Instituto Federal Catarinense - Campus Rio do Sul
Ciência da Computação
Inteligência Artificial
Mathias Artur Schulz

Algoritmo Genético em Python para resolução do Quadrado Mágico

Rio do Sul, 2019

1. Introdução

Um quadrado mágico é uma tabela de lado n, no qual a soma dos números das linhas, das colunas e das duas diagonais são sempre iguais. A respectiva tabela pode ser de qualquer tamanho, no entanto deve ser quadrada e possui valores que não se repetem.

Os quadrados mágicos podem ser classificados em três tipos, que são: Imperfeitos ou defeituosos, não obedecem a todas as regras, como um quadrado mágico em que a soma de todas as linhas e todas as colunas são iguais, mas as diagonais possuem somas diferentes; Hipermágicos, possuem propriedades adicionais, ou seja, obedecem às regras básicas, entretanto a troca de duas colunas de lugar formam outro quadrado mágico; Diabólicos, são quadrados hipermágicos com muitas propriedades ou com propriedades muito complexas, possuem esse nome devido a dificuldade de os formar.

Acredita-se que a origem do quadrado mágico tenha sido na China e na Índia, há cerca de 3000 anos, sendo que os quadrados mágicos ganharam esse nome devido ao fato da crença de que possuíam poderes especiais. Os chineses acreditavam que quem possuísse um quadrado mágico também possuiria sorte e felicidade para toda a vida. Além disso, acreditavam que os quadrados mágicos possuíam símbolos que reuniam os princípios básicos que formavam o universo, no qual os números pares simbolizavam o princípio feminino (Yin), os números ímpares simbolizavam o princípio masculino (Yang), o número 5 representava a Terra, os números 1 e 6 a água, 2 e 7 o fogo, 3 e 8 madeira e 4 e 9 os metais.

No século XV, os quadrados mágicos se tornaram conhecidos na Europa, sendo relacionados com a alquimia e a astrologia. Um quadrado mágico gravado numa placa de prata era usado como amuleto contra a peste, além disso, cada quadrado mágico de ordem 3 até a ordem 9 representava um planeta, que são: Ordem 3 representava Saturno; Ordem 4 representava Júpiter, Ordem 5 representava Marte; Ordem 6 representava Solenoide; Ordem 7 representava Vénus; Ordem 8 representava Mercúrio e ordem 9 representava Luna.

O presente relatório técnico possui como objetivo apresentar a construção de um algoritmo genético desenvolvido na linguagem de programação Python com o propósito de resolver o problema do quadrado mágico. Será abordado e explicado a forma de representação do cromossomo, o fitness e os operadores genéticos, por fim será apresentado o código e os testes realizados.

2. Representação cromossomial

Para a representação de um cromossomo foi utilizado uma matriz construída a partir da função random.choice da biblioteca NumPy, disponível para a linguagem de programação Python.

A matriz quadrada possui ordem n e valores aleatórios que não se repetem. Para geração dos valores aleatórios, sempre será um valor inteiro maior que zero, sendo que o valor máximo permitido será o número de células da matriz ao quadrado, no entanto esse valor pode ser facilmente configurado nas variáveis iniciais do algoritmo. No entanto, o valor máximo a ser gerado não deve ser menor que o número de células da matriz, devido ao fato de não ser permitido valores repetidos.

3. Fitness (Função de Avaliação)

Para geração do fitness de um cromossomo, primeiramente é realizado a soma de cada linha, coluna e das duas diagonais, sendo que cada soma é armazenada em uma posição do array de somas. Com isso, a partir das somas encontradas, é realizado uma média das somas.

Após o cálculo das somas e a média das somas, é realizado uma comparação de cada soma do array de somas com a média e é calculado a diferença entre a respectiva soma e a média. Com isso, o fitness será a soma das diferenças encontradas.

Neste algoritmo, quanto menor o fitness, melhor é o cromossomo. Caso o fitness seja igual a zero, significa que a soma de cada linha, cada coluna e das duas diagonais são iguais, ou seja, é um quadrado mágico.

4. Operadores genéticos

Neste capítulo é apresentado a forma de realização da Mutação e do Crossover para a evolução das gerações.

4.1. Crossover

Para a realização do crossover, primeiramente são selecionados os pais da população, os pais são os melhores cromossomos, a quantidade de pais que serão selecionados podem ser determinados a partir de uma variável inicial do algoritmo.

No processo de crossover é passado por todos os outros cromossomos que não são pais, para cada cromossomo é selecionado aleatoriamente dois pais e é realizado o crossover. Sendo que para cada cromossomo a ser realizado o crossover existe um ponto de corte randômico, o ponto de corte divide o cromossomo atual em duas partes, a primeira parte receberá a parte correspondente do pai 1 e a segunda parte receberá a parte correspondente do pai 2.

4.2. Mutação

Para realização da mutação, primeiramente é determinado uma probabilidade de mutação a partir de uma variável inicial, por padrão determinada com 0.5 de probabilidade de mutação. Com isso, é percorrido todos os cromossomos que não são pais, para cada cromossomo é gerado um número float randômico entre zero e um, caso o número gerado seja menor ou igual a probabilidade de mutação, então ocorrerá a mutação.

Na mutação é gerado um ponto do cromossomo onde ocorrerá a mutação e para essa posição do cromossomo será gerado um novo valor, ocorrendo a mutação no cromossomo. No entanto, o novo valor deve ser diferente de todos os outros presentes no cromossomo.

5. Testes

Quando o algoritmo é executado, inicialmente é apresentando todos os cromossomos da população e então é realizando a evolução das gerações (crossover e mutações). Por fim, é apresentado a última geração obtida, apresentando do pior cromossomo até o melhor cromossomo da geração.

Para cada cromossomo é apresentado o cromossomo obtido, o fitness e o array de somas de cada coluna, linha e as duas diagonais, exemplo com tabela de ordem 3:

– Cromossomo:

```
[[ valor11  valor12  valor13]
 [ valor21  valor22  valor23]
 [ valor31  valor32  valor33]]
```

– Fitness:

```
valor_fitness
```

– Array de somas:

```
[coluna01 coluna02 coluna03 linha01 linha02 linha03
diagonal01 diagonal02]
```

Abaixo é apresentado um exemplo de resultado obtido de um cromossomo após a execução do algoritmo.

```
[[ 2  9  7]
 [13  4  1]
 [ 6  5 10]]

9

[21 18 18 18 18 21 16 17]
```

Diversos testes foram realizados, abaixo é apresentado alguns testes realizados e os resultados obtidos.

Especificações do teste: População de 500 cromossomos, 1000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 3. Em três testes separados, o melhor cromossomo para cada teste foi:

```
Teste 01:
[[ 1 15  6]
 [12  7  3]
 [ 9  0 14]]
2
[22 22 23 22 22 23 22 22]
Tempo de execução: 18.47 segundos.

Teste02:
[[ 4 16  5]
 [ 9  7  6]
 [12  1 13]]
6
[25 24 24 25 22 26 24 24]
Tempo de execução: 19.61 segundos.

Teste 03:
[[14  4 15]
```

```
 [11 12 10]
 [ 7 17  8]]
4
[32 33 33 33 33 32 34 34]
Tempo de execução: 18.39 segundos.
```

Especificações do teste: População de 500 cromossomos, 1000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 5. O melhor cromossomo para o teste foi:

```
Teste 04:
[[10 13 49 24 33]
 [21 31 30 40  8]
 [44 47  1  3 36]
 [22 23 35 43  7]
 [34 16 15 20 45]]
5
[131 130 130 130 129 129 130 131 130 130 130 131]
Tempo de execução: 19.71 segundos.
```

Especificações do teste: População de 500 cromossomos, 1000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 10. O melhor cromossomo para o teste foi:

```
Teste 05:
[[ 84 154 156  89  20  60 131  79 184  98]
 [ 40  78  37 139 170   8 115 194 154 121]
 [137  95 167 174 141  47 175   4 114   2]
 [145  35 104  20  26 157 189 153  55 172]
 [ 22  70  72 187 188 191 136  46  18 126]
 [ 27 173   9 179  86 164  42 160  90 124]
 [192 163 147  24 134   6 109  99 133  49]
 [ 74 127  96  12 159 162  11 151  85 176]
 [158  76 152  51  77 178 144  50  38 129]
 [177  80 116 180  56  82   3 120 185  57]]
59
[1056 1051 1056 1055 1057 1055 1055 1056 1056 1054 1055 1056 1056
1056 1056 1054 1056 1053 1053 1056 1056 1095]
Tempo de execução: 22.19 segundos.
```

Especificações do teste: População de 1000 cromossomos, 1000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 3. O melhor cromossomo para o teste foi:

```
Teste 06:
[[ 9 15  6]
 [ 5 10 13]
 [16  4 11]]
6
[30 29 30 30 28 31 30 32]
Tempo de execução: 39.76 segundos.
```

Especificações do teste: População de 500 cromossomos, 2000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 3. Em três testes separados, o melhor cromossomo para cada teste foi:

```
Teste 07:
[[ 9  1 14]
 [13  8  3]
 [ 2 15  7]]
0
[24 24 24 24 24 24 24 24]
Tempo de execução: 39.75 segundos.

Teste 08:
[[ 7 13 11]
 [15 12  6]
 [ 8  9 14]]
8
[30 34 31 31 33 31 33 31]
Tempo de execução: 39.47 segundos.

Teste 09:
[[ 2  8  7]
 [11  6  1]
 [ 5  4 10]]
2
[18 18 18 17 18 19 18 18]
Tempo de execução: 43.26 segundos.
```

Especificações do teste: População de 1000 cromossomos, 6000 gerações, 2 pais, probabilidade de mutação de 0.5 e tabela de ordem 3. O melhor cromossomo para o teste foi:

```
Teste 10:
[[15  9  7]
 [ 2 11 17]
 [14 10  7]]
5
[31 30 31 31 30 31 33 32]
Tempo de execução: 228.60 segundos / ~3.8 minutos.
```

6. Código documentado

Neste capítulo é apresentado o algoritmo (com comentários explicando o funcionamento de cada método) para resolver o problema do Quadrado Mágico, construído na linguagem de programação Python.

```
1 import time
2 import random
3 import numpy as np
4
```

```
5
6  POPULATION_SIZE = 500
7  GENERATIONS = 1000
8  PARENTS_SIZE = 2
9  MUTATION_PROBABILITY = 0.5
10 TABLE_SIZE = 3
11 CHROMOSOME_SIZE = TABLE_SIZE * TABLE_SIZE
12 MAX_VALUE_TABLE = CHROMOSOME_SIZE + CHROMOSOME_SIZE
13
14
15 # Criacao de um individuo da populacao a partir de uma matriz
      numpy
16 def chromosome():
17     return np.random.choice(
18         MAX_VALUE_TABLE, size=(TABLE_SIZE, TABLE_SIZE), replace=
              False
19     )
20
21
22 # Criacao da populacao
23 def population():
24     return [chromosome() for i in range(POPULATION_SIZE)]
25
26
27 # Metodo que realiza a soma de cada linha, coluna e diagonal do
      cromossomo
28 # Retorna um array com as somas
29 def sumMatrix(chromosome):
30     # Realiza a soma das colunas, cada posicao do array
          representa uma coluna
31     arraySumColumn = np.sum(chromosome, axis=0)
32     # Realiza a soma das linhas, cada posicao do array
          representa uma linha
33     arraySumRow = np.sum(chromosome, axis=1)
34     # Realiza a soma da diagonal principal
35     sumPrimaryDiagonal = np.trace(chromosome)
36     # Realiza a soma da diagonal secundaria a partir da inversao
          das linhas
37     sumSecondaryDiagonal = np.trace(chromosome[::-1])
38     # Concatenacao de todas somas em um unico array
39     chromosomeSum = np.concatenate([
40         arraySumColumn, arraySumRow, [sumPrimaryDiagonal], [
              sumSecondaryDiagonal]
41     ])
42     return chromosomeSum
43
44
45 # Converte cada cromossomo de uma populacao em uma matriz
46 def populationArrayToMatrix(population):
```

```
47      return [i.reshape(TABLE_SIZE, TABLE_SIZE) for i in
            population]
48
49
50  # Converte cada cromossomo de uma populacao em um array
51  def populationMatrixToArray(population):
52      return [i.reshape(-1) for i in population]
53
54
55  # Metodo que calcula o fitness de um chromosome
56  # Fitness: Soma das distancias de cada soma com a media
57  # OBS: Quanto menor o fitness melhor o cromossomo
58  def fitness(chromosome):
59      chromosomeSum = sumMatrix(chromosome)
60      # Realiza a media de todas as somas
61      average = int(sum(chromosomeSum) / len(chromosomeSum))
62
63      fitness = 0
64      for i in range(len(chromosomeSum)):
65          diff = chromosomeSum[i] - average
66          fitness = fitness + (diff if diff > 0 else -diff)
67      return fitness
68
69
70  # Metodo que ordena a populacao de acordo com o fitness
71  def populationSortedByFitness(population):
72      # Monta um array com cada cromossomo e seu respectivo
            fitness
73      populationWithFitness = [(fitness(i), i) for i in population
            ]
74      # Ordena a populacao por fitness - Do pior (maior) fitness
            para o melhor (menor)
75      return [
76          i[1] for i in sorted(populationWithFitness, key=lambda
                chromosome: chromosome[0], reverse=True)
77      ]
78
79
80  # Metodo de selecao dos pais e cruzamento
81  def selectionAndCrossover(population):
82      # Seleciona os pais, cromossomos com melhor fitness (fitness
            mais baixo)
83      parents = population[(len(population) - PARENTS_SIZE):]
84
85      # Passa pelos outros cromossomos realizando o crossover com
            os pais
86      for i in range(len(population) - PARENTS_SIZE):
87          # Caso possua mais de dois pais, e selecionado
                aleatoriamente apenas dois
```

```
88          parents = random.sample(parents, 2)
89
90          # Pega um ponto de corte randomico para realizar o
               crossover
91          cutPoint = random.randint(1, CHROMOSOME_SIZE - 1)
92          # Cromossomo atual recebe do pai 1 os valores antes do
               corte
93          population[i][:cutPoint] = parents[0][:cutPoint]
94          # Cromossomo atual recebe do pai 2 os valores a partir
               do corte
95          population[i][cutPoint:] = parents[1][cutPoint:]
96      return population
97
98
99  # Funcao de mutacao
100 def mutation(population):
101     # Percorre os cromossomos sem contar os pais
102     for i in range(len(population) - PARENTS_SIZE):
103         # Caso o random seja <= a probabilidade de mutacao,
               ocorrera a mutacao
104         if (random.random() <= MUTATION_PROBABILITY):
105             # Posicao que ocorrera a mutacao no cromossomo
106             mutationPoint = random.randint(0, CHROMOSOME_SIZE -
                   1)
107             # Novo valor para mutacao do cromossomo
108             newValue = random.randint(1, MAX_VALUE_TABLE)
109             while(newValue in population[i]):
110                 newValue = random.randint(1, MAX_VALUE_TABLE)
111             population[i][mutationPoint] = newValue
112     return population
113
114
115 # Apresenta cada cromossomo e seu resultado
116 def verification(populationWithFitness):
117     for chromosomeWithFitness in populationWithFitness:
118         print(chromosomeWithFitness[1])
119         print(chromosomeWithFitness[0])
120         arraySum = sumMatrix(chromosomeWithFitness[1])
121         print(arraySum)
122     print('--Resultados: \nChromosome \nFitness')
123     print('Array Sum [Column1 Column2 ColumnN Row1 Row2 RowN
           Diagonal1 Diagonal2] ')
124
125
126 # Codigo principal
127 tempoInicial = time.time()
128 population = population()
129 print('Populacao inicial: ')
130 [print(chromosome) for chromosome in population]
```

```
print('\n'*5)
for i in range(GENERATIONS):
    population = populationSortedByFitness(population)
    population = populationMatrixToArray(population)
    population = selectionAndCrossover(population)
    population = mutation(population)
    population = populationArrayToMatrix(population)
print('\n'*5)
tempoFinal = time.time()


# Resultados
print('Populacao final e resultados: ')
chromosomeAndFitness = [(fitness(i), i) for i in population]
populationWithFitness = [
    i for i in sorted(chromosomeAndFitness, key=lambda
       chromosome: chromosome[0], reverse=True)
]
verification(populationWithFitness)
print('--Tempo de execucao: ')
print(tempoFinal - tempoInicial)
```

7. Considerações Finais

O algoritmo construído na linguagem de programação Python para resolução do Quadrado Mágico apresentou sucesso no crossover e na mutação da população.

O principal fator observado de acordo com os testes para determinar se um cromossomo da geração final é um quadrado mágico são os valores randômicos iniciais, cada execução gera uma geração final com fitness diferentes, devido a aleatoriedade dos cromossomos iniciais. Por exemplo, os testes realizados com uma população de 500 cromossomos e 2000 gerações, o teste 7 gerou um quadrado mágico, no entanto os testes 8 e 9 não geraram.

Quanto maior o número da população e o número de gerações, maior é o tempo de execução. No entanto, nem sempre o fitness será melhor com uma população e o número de gerações maior, como apresentado nos testes. Por exemplo, o melhor cromossomo de uma população de 1000 cromossomos com 6000 gerações apresentou 5 de fitness. Já o pior fitness dos três testes iniciais, com uma população de 500 cromossomos com 1000 gerações apresentou 6 de fitness.

Na maioria das execuções do algoritmo, o melhor cromossomo não é um quadrado mágico, no entanto o cromossomo sempre está bem perto de ser, apresentando um fitness quase igual a zero.