Inteligência Artificial
Problema das Oito Rainhas

Helder Mateus dos Reis Matos[1]

[1]Faculdade de Computação
Instituto de Ciências Exatas e Naturais
Universidade Federal do Pará (UFPA)
Av. Augusto Correa 01, 66075-090 – Belém – PA – Brasil

helder.matos@icen.ufpa.br

Este trabalho descreve a implementação de dois algoritmos de busca local, Stochastic Hill Climbing e Algoritmo Genético, com o intuito de resolver o problema das oito rainhas.

Os códigos-fonte, documentações e quaisquer materiais produzidos nesse projeto encontram-se em https://github.com/hellsdeur/eight-queens-problem.

1. Stochastic Hill Climbing

1.1. Orientações

Foi requerida uma implementação do Stochastic Hill Climbing, onde fossem observados e descritos os seguintes critérios:

- Descrição da função objetivo utilizada na modelagem do problema
- Descrição da codificação utilizada para a solução candidata
- Descrição da(s) heurística(s) e o(s) critérios de parada utilizados pelo algoritmo.
- Execute 50 vezes o algoritmo e calcule: média e desvio padrão do número mínimo de iterações necessário para parar o algoritmo; média e desvio padrão do tempo de execução do algoritmo.
- Construa dois gráficos:
  - 1) plotar a curva com número mínimo de iterações de cada execução.
  - 2) plotar a curva com o tempo de execução do algoritmo de cada execução.
- Mostre, pelo menos, duas soluções distintas encontradas pelo algoritmo.
- Comente e mostre o código fonte do algoritmo desenvolvido.

1.2. Implementação

A implementação do algoritmo foi feita na linguagem Python, pela ampla diversidade de bibliotecas funções, módulos e métodos disponíveis que facilitaram a estrutura dos dados e a legibilidade das instruções. Foram utilizadas as seguintes bibliotecas para a execução do algoritmo.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4 from time import process_time
5 import statistics
```

1.2.1. Codificação da solução candidata

O tabuleiro de xadrez do problema das oito rainhas pode ser facilmente abstraído computacionalmente em um array ou lista bidimensional, onde as casas vazias podem ser representadas por um inteiro 0 e as ocupadas por rainhas representadas por um inteiro 1. Entretanto tal representação armazena muitos dados que não interessam ao algoritmo, onde apenas as posições ocupadas pelas rainhas interessam. Tais representações esparsas devem ser evitadas, a fim de reduzir a complexidade computacional do algoritmo.

Com isso em mente, optou-se por usar uma representação mais condensada do tabuleiro. Uma lista de 8 posições, onde cada índice representa uma coluna, armazena um número que representa a linha, onde esses pares de valores de colunas e linhas são referências para as posições onde as oito rainhas estão. Note que isso também elimina a possibilidade de haver duas rainhas na mesma coluna, melhorando a solução do problema já na geração do mesmo. Dessa forma é possível mapear de maneira muito mais econômica as posições das rainhas.

A função `generate_random_solution()` retorna uma lista com oito listas internas, onde cada lista interna armazena 8 números inteiros gerados aleatoriamente sob uma distribuição uniforme de 0 a 7.

```
7
8  # solução é uma lista de 8 elementos, posição é uma coluna, valor é a linha
9  def generate_random_solution():
10     return [random.randint(0, 7) for i in range(0, 8)]
```

1.2.2. Função Objetivo

Como o objetivo do problema é fazer com que nenhuma rainha se ataque, a função de custo deve tender a diminuir, até um valor ótimo de 0 rainhas atacantes. Assim, a função objetivo foi projetada para que o problema seja de minimização, onde o valor de fitness do algoritmo indica a quantidade de rainhas que estão se atacando.

Duas rainhas estão se atacando quando elas estão presentes na mesma coluna, linha ou diagonal (em qualquer direção). A geração das soluções já eliminou a possibilidade de existirem rainhas na mesma coluna. Dessa forma, a função `calculate_fitness()` conta a quantidade de "colisões"para cada rainha do tabuleiro para a mesma linha (`[22]`) e para as quatro diagonais (`[25,30,35,40]`) e divide o total dessas colisões por 2, obtendo assim a quantidade de pares de rainhas atacantes.

```
12
13 # fitness é a quantidade de pares de rainhas se atacando
14 def calculate_fitness(solution):
15     collisions = 0
16
17     # iterando sobre cada rainha
18     for column in range(0, 8):
19         line = solution[column]
20
21         # contar colisões com a rainha atual na mesma linha
22         collisions += solution.count(line) - 1
23
24         # contar colisões com a rainha atual na diagonal superior esquerda
```

```
        for i, j in zip(range(line-1, -1, -1), range(column-1, -1, -1)):
            if solution[j] == i:
                collisions += 1

        # contar colisões com a rainha atual na diagonal inferior esquerda
        for i, j in zip(range(line+1, 8, +1), range(column-1, -1, -1)):
            if solution[j] == i:
                collisions += 1

        # contar colisões com a rainha atual na diagonal superior direita
        for i, j in zip(range(line-1, -1, -1), range(column+1, 8, +1)):
            if solution[j] == i:
                collisions += 1

        # contar colisões com a rainha atual na diagonal inferior direita
        for i, j in zip(range(line+1, 8, +1), range(column+1, 8, +1)):
            if solution[j] == i:
                collisions += 1

    # metade das colisões = pares de rainhas atacantes
    return collisions//2
```

1.2.3. Heurísticas e Critérios de parada

De posse das funções `generate_random_solution()` e `calculate_fitness()`, o corpo principal do algoritmo Stochastic Hill Climbing pode ser projetado.

Uma variável `max_cln` (*maximum consecutive lateral movements*) indica o número máximo de movimentos laterais consecutivos do algoritmo, funcionando como um critério de parada caso o algoritmo fique preso num mínimo local. Como essa variável deve ser resetada caso o algoritmo saia do mínimo local, esse valor máximo permanece guardado, e `cln` é usado no decorrer do algoritmo. Durante a execução do algoritmo, optou-se por usar $\texttt{cln} = 1000$.

Após inicializar uma solução aleatória e calcular sua respectiva fitness [54:55], um laço é iterado `58:72` até uma de duas condições de parada sejam atendidas:

- Um valor de fitness seja 0, ou seja, onde é alcançado um estado em que nenhuma rainha se ataque.
- O algoritmo esgote sua cota de passos laterais e não encontre uma solução melhor.

Dentro do laço são geradas novas soluções que tem sua fitness comparada com a melhor obtida até então. Se uma solução superar a fitness da melhor solução, ela se torna a melhor. A função retorna uma lista contendo a melhor solução, seu valor de fitness e a quantidade necessária de iterações do laço até que um critério de parada fosse atendido.

```

# max_clm = maximum consecutive lateral movements
def stochastic_hill_climbing(max_clm):
    clm = max_clm # clm variável
    iterations = 0 # número de iterações

    # solução inicial
    best_solution = generate_random_solution()
    best_fitness = calculate_fitness(best_solution)

    # enquanto o fitness mínimo não seja alcançado e ainda restem clms
    while best_fitness > 0 and clm > 0:

```

```
        # gera uma nova solução e calcula o fitness
        new_solution = generate_random_solution()
        new_fitness = calculate_fitness(new_solution)

        # se o novo fitness for menor, ele é tomado como o melhor
        if new_fitness < best_fitness:
            best_solution = new_solution
            best_fitness = new_fitness
            clm = max_clm # reseta o contador de movimentos laterais
        else:
            clm -= 1 # realiza movimento lateral

        iterations += 1

    # retorna a melhor solução, a fitness e o número de iterações até a parada
    return [best_solution, best_fitness, iterations]
```

1.3. Execução e Análise de resultados

1.3.1. Execução

A fim de armazenar alguns dados das 50 execuções pedidas, foram criadas quatro listas `79:82`:

- `solutions`: armazena as soluções das 50 execuções
- `fitnesses`: armazena os valores de fitness das 50 soluções
- `iterations`: armazena o número de iterações necessárias para as 50 execuções
- `runtimes`: armazena o tempo de execução de cada uma das 50 execuções. O cálculo do tempo foi feito através da diferença entre o tempo da CPU antes e depois da execução do Hill Climbing.

```

# execução e análise dos resultados
solutions = [] # soluções para cada uma das 50 execuções
fitnesses = [] # fitness para cada uma das 50 execuções
iterations = [] # quantidades de iterações para cada uma das 50 execuções
runtimes = [] # tempos de execução para cada uma das 50 execuções

# 50 execuções
for i in range(0, 50):
    start = process_time() # tempo da CPU antes da execução
    result = stochastic_hill_climbing(1000) # solução, fitness e iterações
    end = process_time() # tempo da CPU após a execução

    runtimes.append(end - start)
    solutions.append(result[0])
    fitnesses.append(result[1])
    iterations.append(result[2])
```

Foram obtidas as médias e desvios padrão das 50 iterações e tempos de execução.

```

# estatísticas das iterações
iterations_mean = statistics.mean(iterations)
iterations_stdev = statistics.stdev(iterations)
print("Iterations")
print("Mean: ", iterations_mean)
print("StDev: +- ", iterations_stdev)

# estatísticas dos tempos de execução
runtimes_mean = statistics.mean(runtimes)
runtimes_stdev = statistics.stdev(runtimes)
print("\nRuntimes")
print("Mean: ", runtimes_mean, )
print("StDev: +- ", runtimes_stdev)
```

Executando essas instruções, foram obtidos os seguintes dados:

```
Iterations
Mean:  1334.24
StDev: ±  362.21132404344587

Runtimes
Mean:  0.06034100528000003
StDev: ±  0.02321065899734815
```

Figura 1. Média e desvio padrão de iterações e runtimes de 50 execuções

1.3.2. Gráficos

O gráfico das iterações ao longo das execuções foi obtido usando a lista `iterations`:

```
110
111 # plotando gráfico "execução x número de iterações necessárias"
112 x = np.linspace(1, 50, 50)
113 plt.figure(figsize=(20,5))
114 plt.grid()
115 plt.axis([0, 50, 0, max(iterations)])
116 plt.plot(x, iterations)
117 plt.xlabel("Execution")
118 plt.ylabel("Minimum iterations needed")
119 plt.show()
```

Para os mesmos dados usados na figura (1), foi obtida a seguinte plotagem:

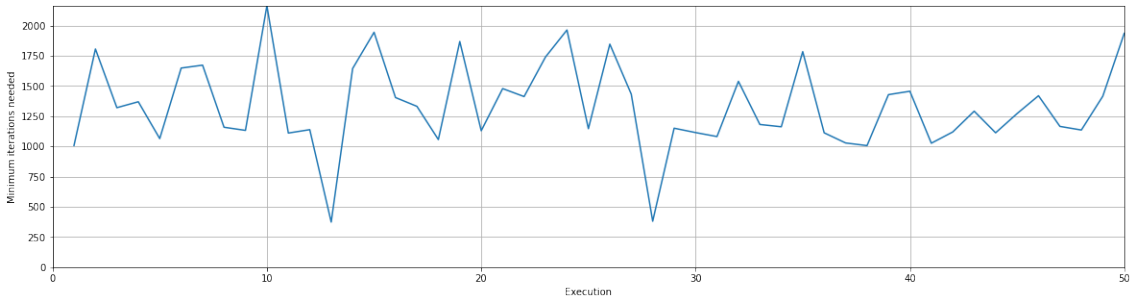

Figura 2. Iterações ao longo das 50 execuções

O gráfico dos tempos de execução ao longo das execuções foi obtido usando a lista `runtimes`:

```
121
122 # plotando gráfico "execução x tempo de execução (em segundos)"
123 x = np.linspace(1, 50, 50)
124 plt.figure(figsize=(20,5))
125 plt.grid()
126 plt.axis([0, 50, 0, max(runtimes)])
127 plt.plot(x, runtimes)
128 plt.xlabel("Execution")
129 plt.ylabel("Runtime (seconds)")
130 plt.show()
```

Para os mesmos dados usados na figura (1), foi obtida a seguinte plotagem:

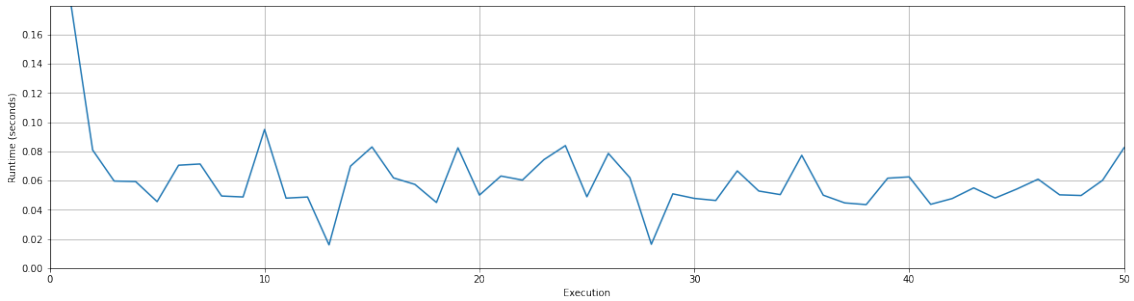

Figura 3. Tempos de execução ao longo das 50 execuções

Podemos notar uma equivalência nas curvas dos dois gráficos, indicando uma relação direta ente a quantidade de iterações e o tempo de execução.

1.3.3. Melhores soluções encontradas

As 5 melhores soluções obtidas para a mesma execução da figura (1) foram separadas e exibidas.

```
# mapeando fitness às respectivas soluções
map_fitness_to_solutions = zip(fitnesses, solutions)

# ordenando as fitness em ordem crescente
sorted_map = sorted(map_fitness_to_solutions)

# obtendo as 5 melhores soluções
top_five_solutions = sorted_map[:5]

# soluções, fitness e representação visual das 5 melhores soluções
for solution in top_five_solutions:
    print('-'*34)
    print("Solution: ", solution[1])
    print("Fitness: ", solution[0])
```

```
Solution:  [4, 2, 0, 6, 1, 7, 5, 3]
Fitness:  0
----------------------------------
Solution:  [5, 2, 0, 6, 4, 7, 1, 3]
Fitness:  0
----------------------------------
Solution:  [0, 2, 5, 7, 4, 1, 3, 6]
Fitness:  1
----------------------------------
Solution:  [1, 5, 4, 6, 3, 0, 2, 7]
Fitness:  1
----------------------------------
Solution:  [2, 4, 6, 1, 3, 6, 0, 7]
Fitness:  1
```

2. Algoritmo Genético

2.1. Orientações

Foi requerida uma implementação de um Algoritmo Genético, onde fossem observados e descritos os seguintes critérios:

- Utilizar obrigatoriamente a codificação binária
- Tamanho da população: 20.
- Seleção dos pais: escolhida pelo(a) projetista.
- Cruzamento: escolhido pelo(a) projetista.
- Taxa de cruzamento: 80%.
- Mutação: escolhida pelo(a) projetista.
- Taxa de mutação: 3%.
- Seleção de sobreviventes: elitista (os melhores indivíduos sempre sobrevivem).
- Critérios de parada: Número máximo de gerações alcançado: 1000; Se a solução ótima for encontrada.
- Apresentar a escolha e explicar o funcionamento dos operadores que foram utilizados: seleção dos pais, cruzamento e mutação.
- Execute 50 vezes o algoritmo e apresente, em forma de tabela, a melhor solução encontrada em cada execução, o valor da função objetivo desta solução encontrada, o tempo de execução e o número da geração em que o algoritmo parou.
- Calcular a média e o desvio padrão do valor da função objetivo do melhor indivíduo, do tempo de execução e o número da geração em que o algoritmo parou (três últimas colunas da tabela).
- Mostre, pelo menos, duas soluções distintas encontradas pelo algoritmo.
- Comente e mostre o código fonte do algoritmo desenvolvido.

2.2. Implementação

Da mesma forma que no Hill Climbing, também foi utilizada a linguagem Python para a codificação deste algoritmo genético. Abaixo as bibliotecas utilizadas:

```
1 import numpy as np
2 import pandas as pd
3 import random
4 from time import process_time
5 import statistics
```

2.2.1. Codificação da solução candidata

Inicialmente, a codificação da solução candidata desse algoritmo é feita da mesma forma que no Hill Climbing, com uma lista contendo oito listas com 8 números aleatórios entre 0 e 7.

```
7
8 # solução é uma lista de 8 elementos, posição é uma coluna, valor é a linha
9 def generate_random_solution():
10     return [random.randint(0, 7) for i in range(0, 8)]
```

Como é necessário o uso da codificação binária para o cruzamento e mutação dos indivíduos, criou-se duas funções de transformação de binário para decimal e vice-versa.

```
12
13 # codificação decimal para codificação binária
14 def to_binary(decimal_solution):
15     return [bin(position)[2:].zfill(3) for position in decimal_solution]
16
17 # codificação binária para codificação decimal
18 def to_decimal(binary_solution):
19     return [int(position, 2) for position in binary_solution]
```

A criação da população do algoritmo depende de um tamanho `k`.

```
21
22 # gerar população de k indivíduos já em binário
23 def generate_population(k):
24     return [to_binary(generate_random_solution()) for i in range(0, k)]
```

2.2.2. Função Objetivo

A função objetivo do Algoritmo Genético também é a mesma implementada no Hill Climbing. Portanto, o problema é configurado como sendo de minimização.

```
26
27 # fitness é a quantidade de pares de rainhas se atacando
28 def calculate_fitness(solution):
29     collisions = 0
30
31     # iterando sobre cada rainha
32     for column in range(0, 8):
33         line = solution[column]
34
35         # contar colisões com a rainha atual na mesma linha
36         collisions += solution.count(line) - 1
37
38         # contar colisões com a rainha atual na diagonal superior esquerda
39         for i, j in zip(range(line-1, -1, -1), range(column-1, -1, -1)):
40             if solution[j] == i:
41                 collisions += 1
42
43         # contar colisões com a rainha atual na diagonal inferior esquerda
44         for i, j in zip(range(line+1, 8, +1), range(column-1, -1, -1)):
45             if solution[j] == i:
46                 collisions += 1
47
48         # contar colisões com a rainha atual na diagonal superior direita
49         for i, j in zip(range(line-1, -1, -1), range(column+1, 8, +1)):
50             if solution[j] == i:
51                 collisions += 1
52
53         # contar colisões com a rainha atual na diagonal inferior direita
54         for i, j in zip(range(line+1, 8, +1), range(column+1, 8, +1)):
55             if solution[j] == i:
56                 collisions += 1
57
58     return collisions//2
```

2.2.3. Operadores

A **seleção dos pais** se deu pela estratégia da roleta. Ao receber como argumentos uma população e a lista de fitness de cada indivíduo, a função `select_parents()` divide a fitness de cada indivíduo pela soma todos os valores de fitness, obtendo assim uma distribuição de probabilidades. `numpy.random.choice` escolhe dois números de acordo

com a distribuição, sem reposição. Assim, os dois números servem como os índices de dois indivíduos na população que são escolhidos como pais da geração atual.

```
60
61 # estratégia da roleta, quanto maior o fitness maior a chance de ser escolhido
62 def select_parents(population, fitnesses):
63     sum_fitnesses = sum(fitnesses)
64     # distr. de prob. para as fitness
65     distribution = [fit/sum_fitnesses for fit in fitnesses]
66
67     # 2 indexes escolhidos dentre os k indivíduos, sem repetição, de acordo com a
       distribuição
68     indexes = np.random.choice(len(population), 2, p=distribution, replace=False)
69
70     return [population[i] for i in indexes]
```

O **cruzamento** dos pais escolhidos depende de uma taxa de cruzamento `cross_rate`. Caso um número aleatório entre 0 e 1 seja menor que `cross_rate` o cruzamento acontece. A troca de material genético é feita através da estratégia do ponto de corte, onde os pais são seccionados em duas partes, gerando filhos por meio da combinação dessas partes. Se o cruzamento não acontecer, os pais se tornam os filhos.

```
72
73 # estratégia do ponto de corte, o crossover tem uma prob. cross_rate de acontecer
74 def crossover(parents, cross_rate):
75     # se houver cruzamento, trocar material genético
76     if random.uniform(0, 1) <= cross_rate:
77         children = []
78
79         # ponto de corte, aleatório entre 1 e 7
80         cutoff = random.randint(1, 7)
81
82         # troca de material genético
83         child_1 = parents[0][0:cutoff] + parents[1][cutoff:]
84         child_2 = parents[1][0:cutoff] + parents[0][cutoff:]
85
86         children.append(child_1)
87         children.append(child_2)
88
89         return children
90
91     return parents # se o cruzamento não for possível, os pais se tornam filhos
```

A **mutação** dos filhos também possui uma probabilidade de acontecer. Caso um número aleatório entre 0 e 1 seja menor que a taxa de mutação `mutate_rate` a mutação é realizada. A estratégia do bit flip consiste em trocar um dos 24 bits que representam a solução pelo seu inverso. Caso a mutação não ocorra, o filho permanece inalterado

```
93
94 # estratégia do bit flip, onde um dos 24 bits de um filho é trocado
95 def mutate(child, mutate_rate):
96     # se a mutação for possível, alterar material genético por bit flip
97     if random.uniform(0, 1) <= mutate_rate:
98         mutated_child = []
99
100        # unificar todo o material genético em uma lista de 24 bits
101        unified_genes = list("".join(child))
102        bit = random.randint(0, 23) # bit a ser alterado
103
104        # troca do bit
105        if unified_genes[bit] == "0":
106            unified_genes[bit] = "1"
107        else:
108            unified_genes[bit] = "0"
109
110        # junta os bits em uma única string
```

```
        mutated_string = "".join(unified_genes)

        # divide os bits de 3 em 3, obtendo a forma original
        for i in range(0, 24, 3):
            mutated_child.append(mutated_string[i:i+3])

        return mutated_child

    return child # se a mutação não for possível, o filho não se altera
```

A seleção dos sobreviventes é feita de forma elitista, onde apenas os `k` indivíduos com a menor fitness sobrevivem para a próxima geração. A fim de ordenar a lista de indivíduos em ordem crescente pela sua respectiva fitness, a função `select_survivors` recebe um mapeamento entre esses dois parâmetros, facilitando a ordenação e o corte.

```
# estratégia elitista, onde os k indivíduos com as menores fitness sobrevivem
def select_survivors(map_fitness_to_individual, k):
    sorted_map = sorted(map_fitness_to_individual) # ordenando as fitness em ordem crescente
    return sorted_map[:k] # retornando os k sobreviventes
```

Com a geração da população em mãos, além dos operadores genéticos, o algoritmo genético foi estruturado para receber os valores de `k` indivíduos da população, a taxa de crossover `cross_rate`, a taxa de mutação `mutate_rate` e o número máximo de gerações `gen`.

Ao iniciar a população original e calcular a fitness dos indivíduos, uma variável `best_solution` passa a armazenar a tupla que representa a solução com a menor fitness.

Em seguida, um laço é executado até que um dos seguintes critérios de parada seja atendido:

- Uma indivíduo com fitness igual a zero seja encontrado.
- O número de gerações passe do que foi estipulado por `gen`.

Dentro do laço são aplicados os operadores genéticos de seleção dos pais, cruzamento, mutação e seleção dos sobreviventes. Ao fim das iterações, a solução com o menor fitness assume o posto de melhor solução. São retornados a melhor solução, sua fitness e a diferença ente o número máximo de gerações e a quantidade de gerações necessárias para para o algoritmo.

```
# tam. da população k, a tx. de crossover cross_rate, a tx. de mutação mutate_rate e máximo de gerações gen
def genetic_algorithm(k, cross_rate, mutate_rate, gen):
    max_gen = gen # usar a original para tirar a diferença
    population = generate_population(k)
    fitnesses = [calculate_fitness(to_decimal(individual)) for individual in population] # fitness para cada indivíduo
    best_solution = min(zip(fitnesses, population)) # 2-tupla que armazena a menor fitness e o melhor indivíduo da população

    while best_solution[0] > 0 and gen > 0:
        # seleção dos pais
        parents = select_parents(population, fitnesses)

        # cruzamento
        children = crossover(parents, cross_rate)

        # mutação e avaliação dos filhos
```

```
        mutated_children = [mutate(child, mutate_rate) for child in children]
        fitnesses_children = [calculate_fitness(to_decimal(individual)) for individual
    in mutated_children]

        # adicionando os filhos na população e suas fitness na lista de fitnesses
        population.extend(mutated_children)
        fitnesses.extend(fitnesses_children)

        # mapeando fitness aos respectivos individuos, k + 2 tuplas
        map_fitness_to_individual = zip(fitnesses, population)

        # selecionando as k melhores tuplas
        survivors = select_survivors(map_fitness_to_individual, k)

        # obtendo a melhor solução até o momento
        best_solution = min(survivors)

        # desfazendo o mapeamento
        fitnesses, population = [list(tup) for tup in zip(*survivors)]

        gen -= 1

    # retorna a melhor solução, sua fitness e a geração de parada
    return [best_solution[1], best_solution[0], max_gen-gen]
```

2.3. Execução e Análise de resultados

2.3.1. Execução

A fim de armazenar alguns dados das 50 execuções pedidas, foram criadas quatro listas `166:172`:

- `execution`: armazena os índices das 50 execuções
- `solutions`: armazena as soluções das 50 execuções
- `fitnesses`: armazena os valores de fitness das 50 soluções
- `generations`: armazena o número de gerações necessárias para as 50 execuções
- `runtimes`: armazena o tempo de execução de cada uma das 50 execuções. O cálculo do tempo foi feito através da diferença entre o tempo da CPU antes e depois da execução do Algoritmo Genético.

```

execution = []
solutions = [] # soluções para cada uma das 50 execuções
fitnesses = [] # fitness para cada uma das 50 execuções
generations = [] # geração de parada para cada uma das 50 execuções
runtimes = [] # tempos de execução para cada uma das 50 execuções


# 50 execuções
for i in range(1, 51):
    execution.append(i) # número da execução
    start = process_time() # tempo da CPU antes da execução
    result = genetic_algorithm(20, 0.8, 0.03, 1000) # solução, fitness e geração de
    parada
    end = process_time() # tempo da CPU após a execução

    runtimes.append((end - start)*1000)
    solutions.append(result[0])
    fitnesses.append(result[1])
    generations.append(result[2])
```

Executando essas instruções, foram obtidos os seguintes dados:

```
Fitnesses
Mean:  1.44
StDev: ±  0.5771145679064045

Runtimes
Mean:  211.01480349999994
StDev: ±  49.443398489605265

Generations
Mean:  966.42
StDev: ±  167.69774681167988
```

Figura 4. Média e desvio padrão de iterações e runtimes de 50 execuções

O seguinte dataframe organiza as informações obtidas em um formato de tabela.

```
187
188 df = pd.DataFrame({ "Runtime": execution,
189                     "Best Solution": solutions,
190                     "Fitness": fitnesses,
191                     "Runtime (ms)": runtimes,
192                     "Generation": generations})
```

	Runtime	Best Solution	Fitness	Runtime (ms)	Generation
1	[000, 101, 101, 010, 110, 011, 111, 100]	1	297.834493	1000
2	[100, 001, 101, 000, 010, 110, 000, 011]	2	212.572509	1000
3	[101, 010, 001, 110, 000, 011, 000, 100]	2	211.935033	1000
4	[010, 101, 001, 100, 000, 011, 110, 010]	1	208.666921	1000
5	[001, 011, 111, 010, 100, 010, 000, 101]	1	247.128773	1000
6	[000, 011, 000, 111, 101, 001, 001, 100]	2	212.175686	1000
7	[100, 000, 011, 101, 111, 001, 110, 010]	0	57.700558	272
8	[000, 010, 100, 001, 111, 000, 011, 001]	2	222.602386	1000
9	[001, 101, 000, 010, 100, 111, 011, 110]	1	223.381591	1000
10	[010, 100, 010, 111, 101, 011, 000, 110]	1	201.594647	1000
11	[010, 110, 001, 011, 111, 000, 011, 101]	1	197.965878	1000
12	[101, 010, 000, 111, 011, 001, 110, 010]	1	341.667491	1000
13	[010, 000, 101, 001, 001, 110, 000, 011]	2	340.293298	1000
14	[100, 000, 000, 011, 001, 110, 010, 010]	2	272.134746	1000
15	[100, 001, 011, 000, 010, 111, 101, 000]	1	269.122716	1000
16	[000, 000, 110, 010, 101, 111, 001, 011]	2	226.783622	1000
17	[000, 011, 101, 111, 001, 110, 000, 010]	1	201.621874	1000
18	[000, 110, 001, 101, 000, 011, 111, 010]	2	202.148738	1000
19	[010, 101, 001, 100, 111, 000, 110, 011]	0	10.997169	49
20	[010, 111, 001, 001, 101, 000, 110, 011]	1	200.171643	1000
21	[100, 001, 011, 000, 111, 101, 010, 110]	1	216.194762	1000
22	[010, 101, 010, 110, 001, 011, 111, 000]	1	206.083222	1000
23	[010, 111, 101, 000, 001, 100, 110, 011]	1	209.821751	1000
24	[101, 000, 000, 011, 111, 010, 100, 001]	2	204.371913	1000
25	[011, 000, 100, 001, 001, 101, 111, 010]	1	200.411869	1000

(a) 25 primeiras execuções

25	[011, 000, 100, 001, 001, 101, 111, 010]	1	200.411869	1000
26	[111, 011, 110, 011, 001, 100, 111, 101]	2	204.467877	1000
27	[100, 110, 001, 010, 101, 011, 000, 100]	2	198.492120	1000
28	[000, 100, 111, 000, 010, 110, 001, 011]	1	202.191685	1000
29	[001, 100, 000, 110, 000, 010, 101, 111]	2	203.480669	1000
30	[110, 000, 111, 001, 100, 010, 101, 011]	1	197.244086	1000
31	[011, 110, 000, 101, 001, 001, 111, 010]	1	203.653422	1000
32	[100, 001, 101, 010, 110, 011, 111, 000]	1	204.995169	1000
33	[000, 010, 101, 111, 001, 011, 000, 110]	1	206.122138	1000
34	[000, 011, 101, 010, 110, 001, 111, 100]	1	201.015202	1000
35	[010, 110, 001, 110, 100, 000, 000, 011]	2	200.459118	1000
36	[011, 001, 100, 001, 101, 000, 010, 000]	2	205.244232	1000
37	[100, 000, 111, 101, 010, 101, 001, 011]	2	201.663364	1000
38	[101, 000, 110, 001, 101, 111, 000, 100]	2	199.253065	1000
39	[001, 111, 000, 011, 110, 010, 101, 010]	1	197.661253	1000
40	[101, 001, 110, 010, 000, 111, 000, 100]	2	198.786250	1000
41	[001, 101, 010, 000, 111, 000, 010, 110]	2	208.035353	1000
42	[101, 010, 000, 111, 011, 001, 110, 010]	1	200.371685	1000
43	[010, 100, 001, 001, 011, 110, 000, 111]	2	207.141485	1000
44	[010, 000, 011, 110, 000, 101, 001, 100]	1	200.781931	1000
45	[001, 101, 000, 110, 011, 111, 010, 111]	1	200.357468	1000
46	[000, 101, 000, 010, 111, 100, 001, 011]	2	201.783460	1000
47	[001, 011, 110, 000, 111, 000, 100, 101]	2	204.347263	1000
48	[100, 000, 101, 000, 110, 001, 111, 001]	2	208.764754	1000
49	[000, 110, 001, 101, 111, 001, 011, 010]	2	236.341560	1000
50	[111, 100, 000, 000, 101, 001, 110, 001]	2	262.702277	1000

(b) 25 últimas execuções

2.3.2. Melhores soluções encontradas

As 5 melhores soluções obtidas para a mesma execução da figura (4) foram separadas e exibidas.

```
194
195 # 5 melhores soluções encontradas
196
197 # mapeando fitness às respectivas soluções
198 map_fitness_to_solutions = zip(fitnesses, solutions)
199
200 # ordenando as fitness em ordem crescente
```

```python
sorted_map = sorted(map_fitness_to_solutions)

# obtendo as 5 melhores soluções
top_five_solutions = sorted_map[:5]

for solution in top_five_solutions:
    print('-'*34)
    print("Solution: ", solution[1])
    print("Fitness: ", solution[0])
```

```
----------------------------------
Solution:  ['010', '101', '001', '100', '111', '000', '110', '011']
Fitness:  0
----------------------------------
Solution:  ['100', '000', '011', '101', '111', '001', '110', '010']
Fitness:  0
----------------------------------
Solution:  ['000', '010', '101', '111', '001', '011', '000', '110']
Fitness:  1
----------------------------------
Solution:  ['000', '011', '101', '010', '110', '001', '111', '100']
Fitness:  1
----------------------------------
Solution:  ['000', '011', '101', '111', '001', '110', '000', '010']
Fitness:  1
```