

INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ELIMAR MACENA
LAVÍNIA CORTELETTI

Otimização por Enxame de Partículas

Serra - ES
2021

O Algoritmo

O algoritmo de otimização por enxame de partículas tem como origem o comportamento de seres vivos no próprio ambiente, mais especificamente o comportamento de bando de certos animais.

Assim como os pássaros que voam em bando, ao encontrar uma boa fonte de alimentos a informação se propaga, o presente algoritmo busca simular tal comportamento ao se aproximar de uma resposta ótima.

Segundo Craig Reynolds, esse comportamento de bando tem como origem de 3 ações básicas: a separação, o alinhamento e a coesão.

O algoritmo em si foi desenvolvido por James Kennedy e Russel Eberhart em 1995 e está fortemente ligado com o mundo real através de suas analogias, a terminologia *partícula* visa trazer a mesma ideia que pássaro, o *enxame* faz ligação com o bando de animais e assim por diante.

A transmissão de informação entre as partículas que compõem um enxame ocorre através de dois modos, influência social, onde uma única partícula consegue interferir nas demais (nesse caso, a partícula com melhor resultado) e também temos a influência individual, onde o conhecimento de cada partícula opera sobre ela mesmo.

As partículas de um enxame devem trabalhar em um espaço finito, onde a movimentação é atualizada a cada iteração. A movimentação de uma partícula pode sofrer diferentes influências, como é proposto na literatura.

Problema Proposto

Para o presente trabalho foi proposto um problema de minimização de uma função. Dado a função na Figura 1, o enxame de partícula deve encontrar x e y que resulte no menor valor final da função.

$$f(x, y) = -(y + 47) * \sin \sqrt{\left| \frac{x}{2} + (y + 47) \right|} - x * \sin \sqrt{|x + (y + 47)|}$$

Figura 1

O intervalo utilizado por x e y deve ser [-512,+512] e a velocidade de cada partícula deve estar contida no intervalo [-77,+77], intervalo esse que representa 15% do espaço dado.

Solução

Linguagem de Programação

Para a solução desse problema foi utilizada a linguagem de programação Python, versão 3.8.7, a escolha é dada pela praticidade de uso da linguagem.

Execução do Código

Dados como população, intervalo de interação e informações do gênero foram inseridas em um arquivo de constantes (`src/common/constants.py`), sendo assim a alteração de ambiente de problema é necessário alterar apenas os valores neste documento. Por conta disso é necessário apenas executar o arquivo principal de maneira padrão para que o processo ocorra de maneira correta.

Alteração de Parâmetros

Como o programa centraliza os parâmetros utilizados em um arquivo de constantes, para a alteração de tais parâmetros é necessário apenas acessar o arquivo `src/common/constants.py` para que os valores sejam alterados. Em adição, também é utilizado um valor de seed no arquivo, isso possibilita a repetição de cenários aleatórios em diferentes ambientes e facilitar a verificação de possíveis erros de código.

Para que a criação de arquivos ocorra de maneira esperada, abra o terminal(ou cmd) de dentro da pasta `psa_minimizer` e execute o comando a seguir

```
python ./src/main.py
```

Saída Esperada

Como saída temos dois tipos de arquivos, um arquivo com informações globais e um com informações mais focadas na população e máximo de iteração de um enxame.

No arquivo global temos um arquivo csv com as seguintes informações: população, máximo de população, execução, melhor fitness, média dos resultados e por fim o desvio padrão, todas essas informações compõem uma linha, um exemplo de saída é mostrado na Figura 2. A nomenclatura seguida para essa saída é `executions statistic.csv`

| population | max_iterations | execution | best_fitness | mean | standard_deviation |
|------------|----------------|-----------|--------------|-----------|--------------------|
| 50 | 20 | 0 | -885.9867 | -835.3511 | 101.1132 |
| 50 | 20 | 1 | -718.1669 | -713.2763 | 14.6919 |
| 50 | 20 | 2 | -716.6355 | -702.0102 | 52.3835 |
| 50 | 20 | 3 | -716.3943 | -704.4911 | 30.1599 |
| 50 | 20 | 4 | -959.5169 | -924.7514 | 74.5199 |
| 50 | 20 | 5 | -894.2669 | -869.4058 | 48.2400 |
| 50 | 20 | 6 | -888.9435 | -870.5969 | 57.4173 |
| 50 | 20 | 7 | -935.1316 | -933.4822 | 1.8253 |
| 50 | 20 | 8 | -786.4399 | -777.1279 | 17.4810 |
| 50 | 20 | 9 | -713.0117 | -689.7225 | 41.9853 |

Figura 2.

A segunda saída esperada se trata de um conjunto de arquivo, contém a média de cada iteração no conjunto total de execuções. Com os parâmetros inicialmente solicitados, é esperado 6 arquivos csv como saída, todos seguindo a nomenclatura `executions_population{população}_iterationsmax{total de iteraçao}.csv`. O resultado esperado é mostrado na Figura 3 a seguir.

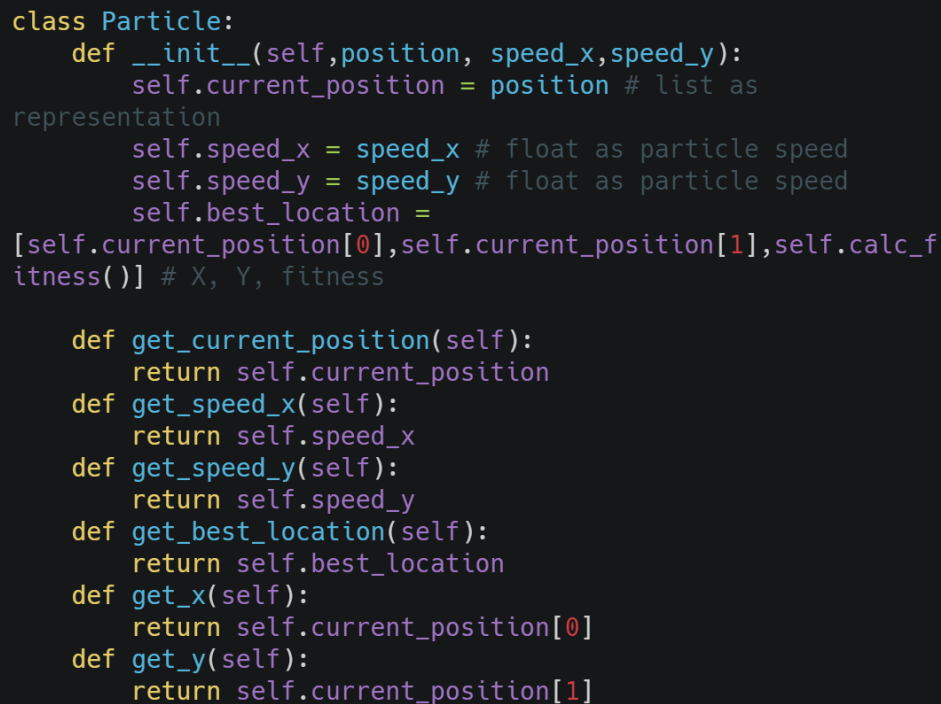
| iteration | mean_execution |
|-----------|--------------------|
| 0 | -649.9785440891510 |
| 1 | -743.3861950318874 |
| 2 | -777.9411248721902 |
| 3 | -795.4100052103587 |
| 4 | -802.3502776026328 |
| 5 | -810.8037349712824 |
| 6 | -812.4906482890171 |
| 7 | -814.8233095735966 |
| 8 | -815.2601016161631 |
| 9 | -815.9796569731295 |
| 10 | -817.7488423658812 |
| 11 | -819.2246321284807 |
| 12 | -819.6434413370101 |
| 13 | -819.9107231304761 |
| 14 | -820.3631614734461 |
| 15 | -820.5042055401657 |
| 16 | -820.5800318089998 |
| 17 | -821.2615218857338 |
| 18 | -821.3211107982555 |
| 19 | -821.4493918813148 |

Figura 3.

Implementação

Partícula (particle)

A fim de representar a partícula utilizada no algoritmo, foi criado um objeto contendo as informações e funções necessárias para o desenvolvimento completo do algoritmo. Na Figura 4 temos a implementação básica do objeto.



```
class Particle:
    def __init__(self, position, speed_x, speed_y):
        self.current_position = position # list as
        representation
        self.speed_x = speed_x # float as particle speed
        self.speed_y = speed_y # float as particle speed
        self.best_location =
        [self.current_position[0], self.current_position[1], self.calc_f
        itness()] # X, Y, fitness

    def get_current_position(self):
        return self.current_position
    def get_speed_x(self):
        return self.speed_x
    def get_speed_y(self):
        return self.speed_y
    def get_best_location(self):
        return self.best_location
    def get_x(self):
        return self.current_position[0]
    def get_y(self):
        return self.current_position[1]
```

Figura 4.

Para que seja realizada a movimentação de um particular, é necessário a definição dos seus valores de velocidade nos dois eixos, x e y. A atualização de velocidade ocorre pelo método `new_speed_calc`, sua implementação é mostrada na Figura 5. No código implementado foi feito o uso de ponderação de inércia, o valor W , a escolha foi feita devido aos bons resultados alcançados em diferentes problemas. Para que seja possível calcular a nova velocidade de uma partícula temos três pontos principais, em nosso caso, o valor W , a melhor partícula global (influência social) e a melhor partícula local (influência individual), à medida que os melhores valores se aproximam, menor é velocidade necessária para a partícula.

```

def new_speed_calc(self, w_value:float, best_particle,
is_x:bool):
    # As a bidimensional space, the use of IF ELSE is
    enough
    new_speed = 0.0
    if is_x:
        local_calc = random() * (self.best_location[0] -
self.current_position[0])
        global_calc = random() * (best_particle.get_x() -
self.current_position[0])
        new_speed = (w_value * self.speed_x) + (const.C1 *
local_calc) + (const.C2 * global_calc)
        # end IF is_x
    else:
        local_calc = random() * (self.best_location[1] -
self.current_position[1])
        global_calc = random() * (best_particle.get_y() -
self.current_position[1])
        new_speed = (w_value * self.speed_y) + (const.C1 *
local_calc) + (const.C2 * global_calc)
        # end ELSE
    if new_speed > const.MAX_SPEED:
        new_speed = const.MAX_SPEED
        return new_speed
    if new_speed < const.MIN_SPEED:
        new_speed = const.MIN_SPEED
        return new_speed
    return new_speed

```

Figura 5.

A função fitness utilizada no problema nada mais é do que a implementação da função que o problema se propõe em minimizar. Sua implementação pode ser vista na Figura 6

```

def calc_fitness(self):
    x = self.current_position[0]
    y = self.current_position[1]
    y_operation = y + 47

    sin1_content = abs( (x/2) + y_operation)
    sin1 = math.sin(math.sqrt(sin1_content))
    term1 = -(y_operation) * sin1

    sin2_content = abs(x - y_operation)
    sin2 = math.sin(math.sqrt(sin2_content))
    term2 = -x * sin2
    final_result = term1 + term2
    return final_result

```

Figura 6.

Enxame (Swarm)

Assim como na construção da partícula, o objeto Swarm visa transpor as informações necessárias do enxame para a execução do código. A implementação base do objeto Swarm pode ser vista na Figura 7, nela temos dois atributos, `population_control`, contendo a informação de todas as partículas que compõem o enxame, e `best_particle`, a melhor partícula de todo enxame até o momento.

```
class Swarm:
    def __init__(self, population_size):
        self.population_control = [None] * population_size
        self.best_particle = None
        self.__create_initial_population()

    def get_population_control(self):
        return self.population_control
    def get_best_particle(self):
        return self.best_particle
```

Figura 7

Como pode ser visto na figura, assim que um enxame é criado, uma população aleatória para o enxame é criada. A criação da população ocorre de maneira simples, é gerada uma velocidade x e y inicial que será repassada para todas as partículas que serão criadas, após isso é definida uma posição aleatória no plano e então armazenada a partícula no `population_control` do objeto. A criação da população é exibida na Figura 8.

```
def __create_initial_population(self):
    counter = 0;
    initial_speed_x = util.create_initial_speed() #initial speed used to
the whole swarm
    initial_speed_y = util.create_initial_speed() #initial speed used to
the whole swarm
    while counter < len(self.population_control):
        new_particle = Particle(util.create_initial_location(),
initial_speed_x, initial_speed_y)
        self.population_control[counter] = new_particle
        counter += 1
    # end WHILE counter
    # set any particle as contructor best.
    self.best_particle = self.population_control[0]
```

Figura 8.

Como citado anteriormente, na solução proposta, foi feito o uso de redução linear da ponderação de inércia, referenciada nas fórmulas como W . Como é feito o uso de tal informação foi necessário transpor a fórmula exibida na Figura 9 para código.

$$w^{k+1} = x_{max} - k \left(\frac{w_{max} - w_{min}}{k_{max}} \right)$$

Figura 9.

A sua implementação foi realizada no objeto Swarm, dado que é algo que extrapola o conhecimento de uma partícula, pois trata de informações de máximo de iterações e de iteração corrente. Sua implementação é exibida na Figura 10

```
def get_w_value(self, current_iteration, max_iteration):
    w_fraction = (const.W_MAX - const.W_MIN) / max_iteration
    w_value = const.W_MAX - (current_iteration * w_fraction)
    return w_value
```

Figura 10.

Os valores atribuídos para W seguem os valores sugeridos pela literatura, wmax = 0,90 e wmin = 0,40. Esses valores foram mantidos devido aos bons resultados encontrados em problemas de gêneros distintos.

Resultados

Como avaliação final dos dados alcançados, vale o comentário de dois pontos, as estatísticas globais das execuções e também o comportamento encontrado nas execuções analisadas de maneira mais fechada.

Numa visão global, as estatísticas nos mostram os seguintes dados: populações menores apresentam um desvio padrão muito alto, suas partículas de baixo resultado tendem a ficar mais afastadas de modo geral, mesmo com o incremento de execuções esse comportamento tende a permanecer.

A Figura 11 tem como objetivo ilustrar o comportamento citado, nela temos as estatísticas de um enxame de população de 50 partículas e 20 iterações, em suas 10 execuções o seu desvio padrão não pareceu estável. Mais dados relacionados a estatísticas globais podem ser vistas no arquivo `outputs/executions_statistic.csv`.

| population | max_iterations | execution | best_fitness | mean | standard_deviation |
|------------|----------------|-----------|--------------|-----------|--------------------|
| 50 | 20 | 0 | -885.9867 | -835.3511 | 101.1132 |
| 50 | 20 | 1 | -718.1669 | -713.2763 | 14.6919 |
| 50 | 20 | 2 | -716.6355 | -702.0102 | 52.3835 |
| 50 | 20 | 3 | -716.3943 | -704.4911 | 30.1599 |
| 50 | 20 | 4 | -959.5169 | -924.7514 | 74.5199 |
| 50 | 20 | 5 | -894.2669 | -869.4058 | 48.2400 |
| 50 | 20 | 6 | -888.9435 | -870.5969 | 57.4173 |
| 50 | 20 | 7 | -935.1316 | -933.4822 | 1.8253 |
| 50 | 20 | 8 | -786.4399 | -777.1279 | 17.4810 |
| 50 | 20 | 9 | -713.0117 | -689.7225 | 41.9853 |

Figura 11.

Ao falarmos de resultados mais específicos, foram gerados 6 gráficos para a análise de como cada padrão de enxame se comportou em suas iterações. Os gráficos que exibidos a seguir foram criados com base nas saídas `executions_population{população}_iterationsmax{total de iteracao}.csv` em um programa externo. Cada um desses arquivos apresenta a média do melhor valor nas 10 execuções solicitadas para cada padrão de enxame.

População 50, Num. Iterações 20

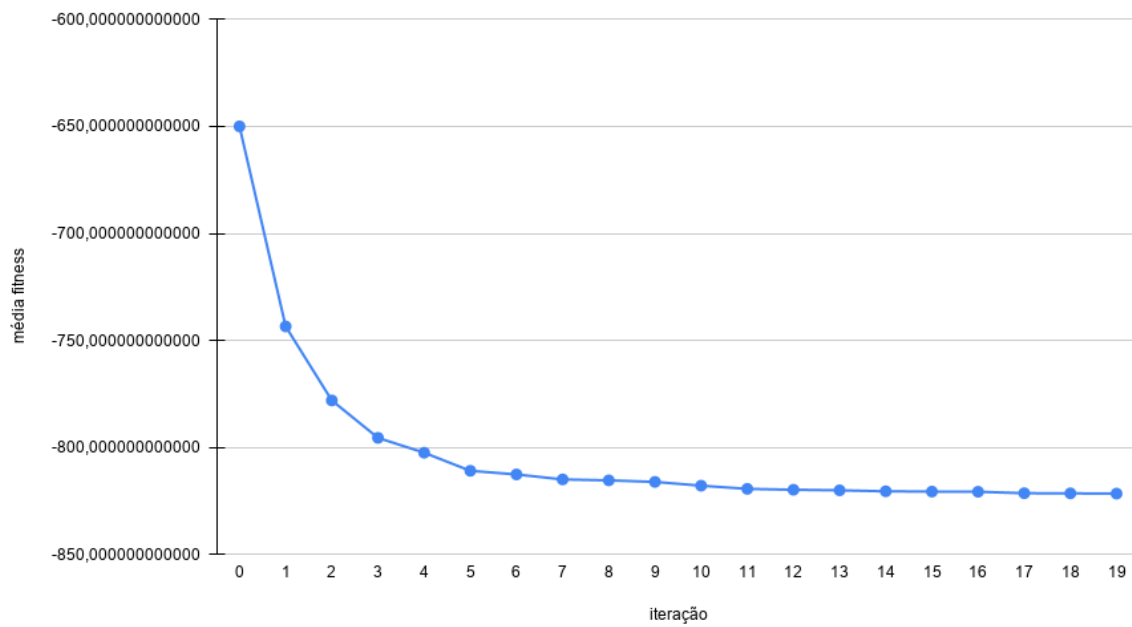


Figura 12.

População 50, Num. Iterações 50

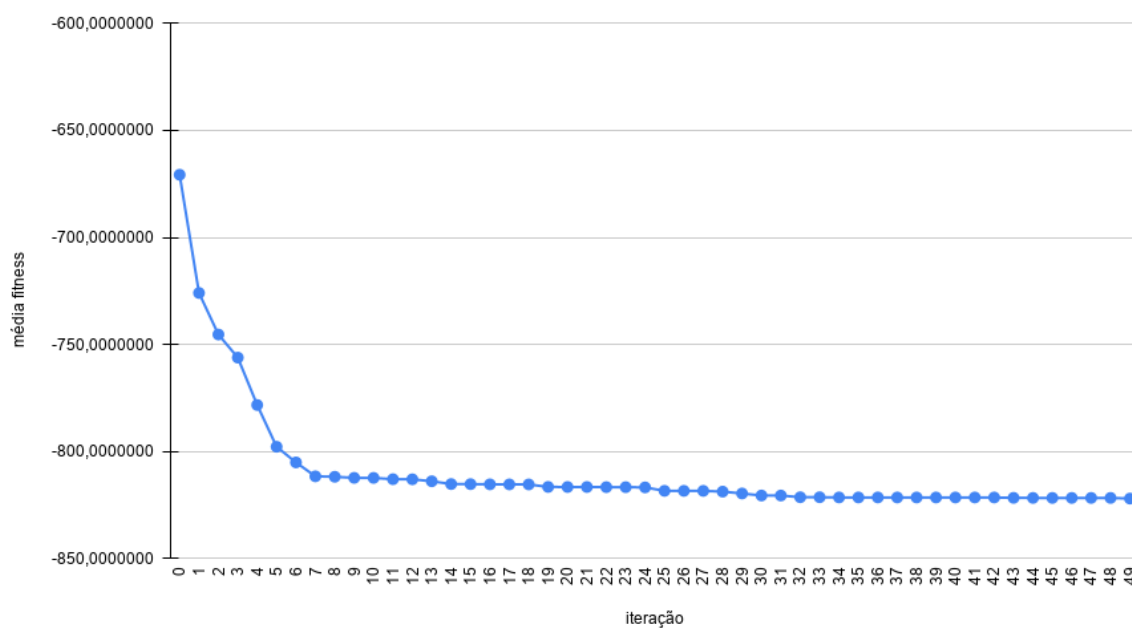


Figura 13.

População 50, Num. Iterações 100

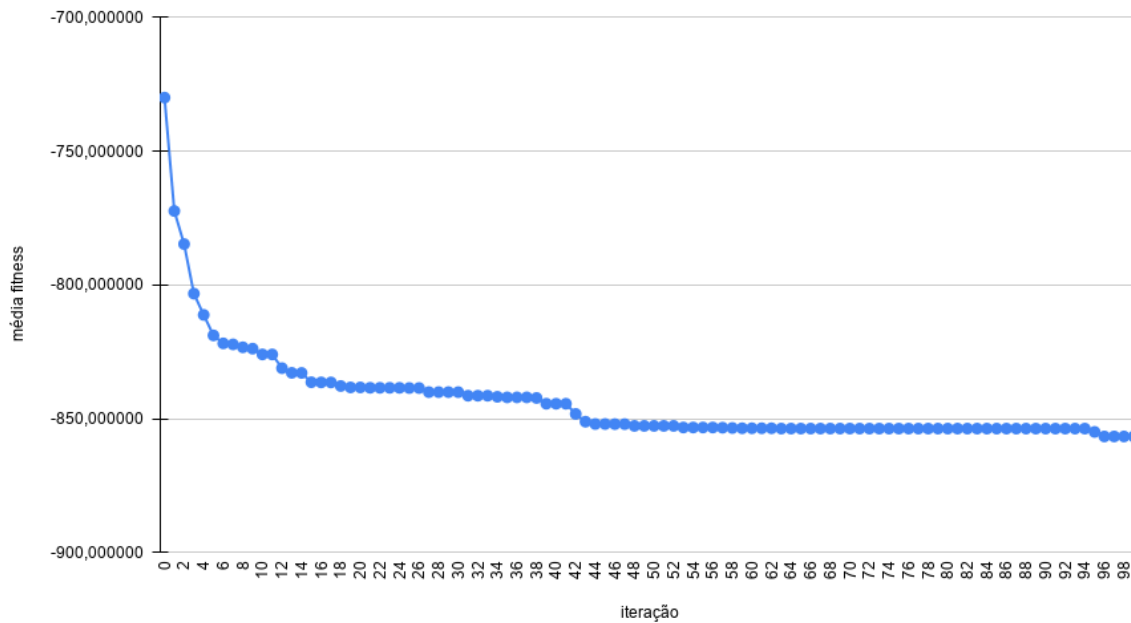


Figura 14.

Ao fazermos uma breve análise nas Figuras 12,13 e 14 vemos que até a 9 iteração temos um certo ganho nos resultados e então encontramos uma certa estabilidade nos melhores resultados, porém na Figura 14, onde temos o maior número de iteração, temos dois novos picos de ganho, iteração 41 e 94, após e anteriormente a essas iterações temos ganhos com pouca representatividade.

População 100, Num. Iterações 20

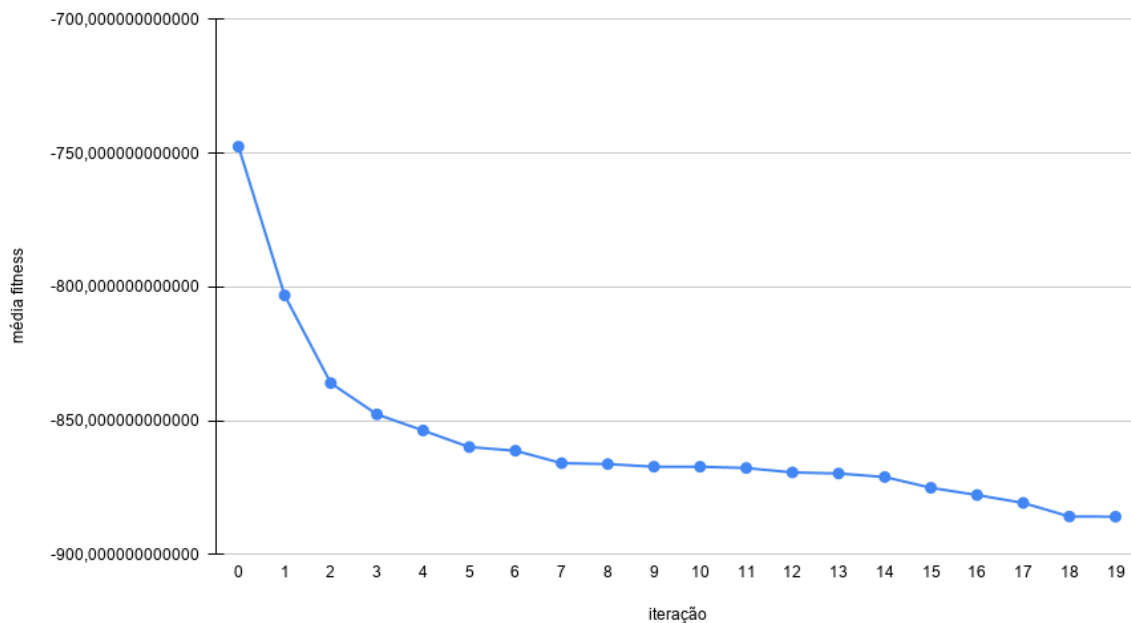


Figura 15.

População 100, Num. Iterações 50

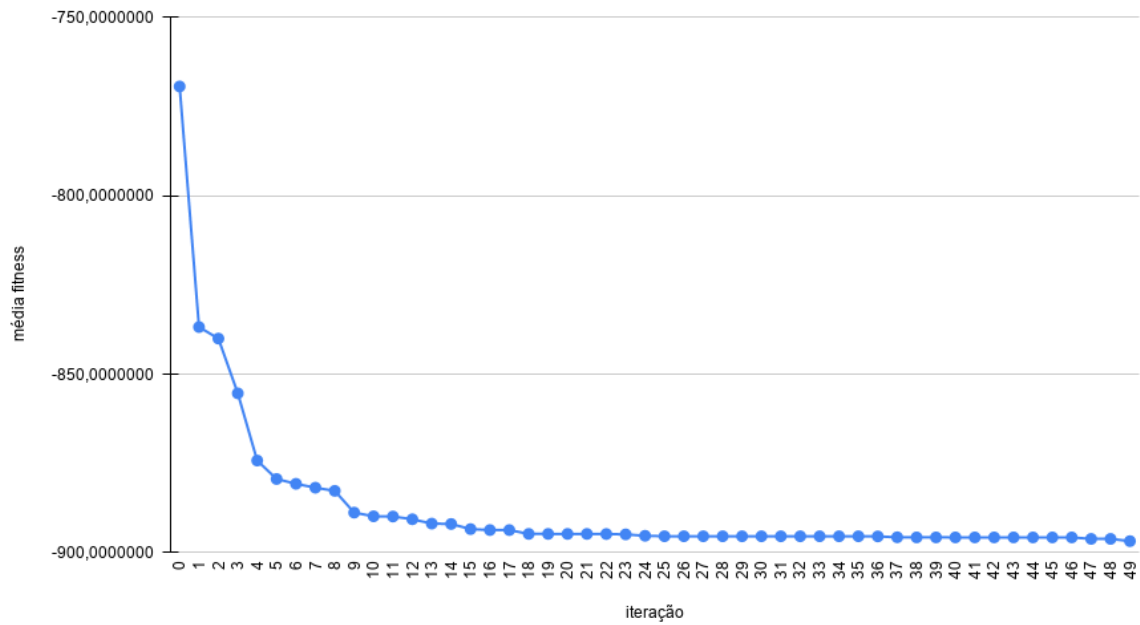


Figura 16.

População 100, Num. Iterações 100

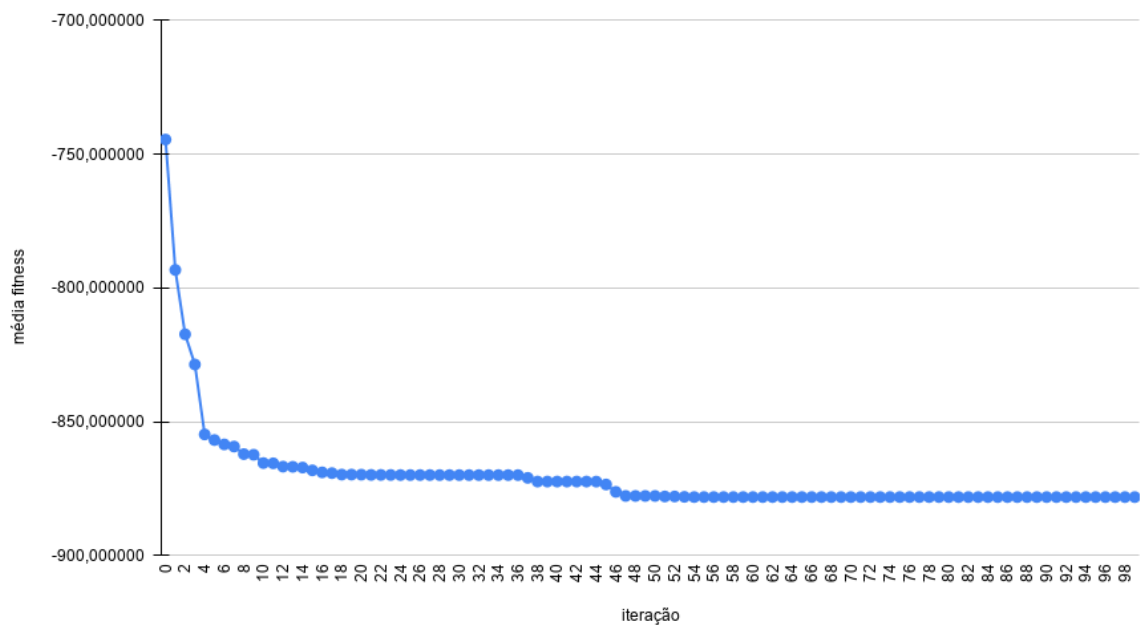


Figura 17.

Nas Figuras 15, 16 e 17 temos o uso de 100 partículas no enxame, e como nos experimentos de 50 partículas, temos uma estabilização grande para depois que depois ocorra novamente um ganho, porém os pontos de picos foram alcançados mais rapidamente.

Analizando os diferentes padrões de definição do enxame, a combinação que traz resultados mais plausíveis e com uma possível redução de custo, seria a combinação de 100 partículas com 50 iterações, pois o conjunto foi capaz de achar um fitness melhor do que a utilização de 50 partículas e 100 iterações e também obteve um resultado próximo ao seu irmão, utilizando 100 partículas e 100 iterações.