INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHAREL EM SISTEMAS DE INFORMAÇÃO

DAVID DE MOURA MARQUES

ENXAME DE PARTÍCULAS

Inteligência Artificial

SERRA
2021

DAVID DE MOURA MARQUES

ENXAME DE PARTÍCULAS

Inteligência Artificial

Trabalho apresentado no curso de graduação do Instituto Federal do Espírito Santo.

Orientador: M.e Eduardo Max Amaro Amaral

SERRA

2021

Sumário

Introdução

A área da inteligência artificial (IA) é uma área que vem crescendo muito nos ultimos anos com cada vez mais destaque devido as novas necessidades de mercado, entretanto este ramo possui diversas segmentações com abordagens diferentes afim de resolver problemas de todos os generos.
Neste relatório iremos tratar de uma abordagem conhecida como enxame de partículas, ou, do inglês *particle swarm optimization* ou **PSO**. Este ramo da IA busca por otimizações de problemas através de iterações avaliando-se os candidatos com base em uma dada medida de qualidade. É classificado por alguns autores como sendo um ramo da computação evolucionária, por basear-se em um movimento natural de enxames, porém, por não ter aspectos como mutação, seleção e combinação, alguns o classificam de forma diferente, muitas vezes como um ramo da famila da swarm intelligence.
A seguir explicaremos como funciona em teoria o algoritmo de enxame de partículas de uma forma bem superficial, após isso será explicado o problema proposto que será resolvido através deste algoritmo, em sequencia será apresentado o código desenvolvido baseado na teoria do algoritmo onde será apresentada a solução do problema proposto, por fim análises estatísticas dos resultados obtidos serão apresentadas.

Enxame de Partículas

Proposto em 1995 por Kennedy e Eberhart o algoritmo de enxame de partículas basea-se nos movimentos feitos por enxames de insetos, como abelhas, vespas e etc. observou-se que nestes movimentos haviam líderes que apontavam a direção para o grupo a medida que se aproximavam do objetivo do enxame, viu-se também que este líder não era um individuo fixo, ele podia se alterar, caso outro agente estivesse mais próximo do objetivo do grupo, então este tornava-se o lider e passava a indicar o caminho, esta revisão de quem estaria mais próximo do objetivo era feita inumeras vezes e, por consequencia, o lider também era alterado inúmeras vezes, desta forma todo o grupo se ajudava e a maioria absoluta dos membros alcançava o objetivo final.

Baseado nisso surgiu uma teoria computacional que propõe que lancemos indivíduos aleatórios no espaço que queremos analizar, o conjunto destes indivíduos será o enxame, então através de uma função que avalie qual indivíduo está mais próximo do objetivo, nós encontramos o lider do enxame. Além disso, sabendo o quão bem posicionado está um indivíduo e quem é o lider do bando, podemos calcular uma velocidade de movimento para aquele indíviduo e em seguida reposicioná-lo em nosso espaço de busca movimentando-o de acordo com sua nova velocidade. É importante também, avaliarmos e armazenarmos a melhor posição em que aquele individuo em questão já esteve.

A função que calcula o quão próximo um agente está em relação ao objetivo vai variar de acordo com o problema a ser resolvido, já a função de velocidade deve ser calculada utilizando a seguinte fórmula:

Velocidade = (pBest - pAtual) * $\boldsymbol{\phi}1$ + (gBest - pAtual) * $\boldsymbol{\phi}2$

Onde,

pBest: É a melhor posição que o indivíduo já esteve até o momento

pAtual: É a posição atual do indivíduo
gBest: É a posicção do melhor indivíduo do grupo, o líder.

$\boldsymbol{\phi}1$ = random(0.0, 1.0) * C1, onde C1 é uma constante qualquer
$\boldsymbol{\phi}2$ = random(0.0, 1.0) * C2, onde C2 é uma constante qualquer

Obs: $\boldsymbol{\phi}1$ e $\boldsymbol{\phi}2$ são muito parecidos, variando apenas pela constante que deverá ser definida pelo programador no momento da implementação afim de dar maior peso para um ou para outro

Em resumo, os passos para implementação do algoritmo devem ser os seguintes:

1. Determine o número de partículas P da população.
2. Inicialize aleatoriamente a posição inicial (x) de cada partícula p de P.
3. Atribua uma velocidade inicial (v) igual para todas as partículas.
4. Para cada partícula p em P faça:
    (a) Calcule sua aptidão fp = f (p).
    (b) Calcule a melhor posição da partícula p até o momento (pB).
5. Descubra a partícula com a melhor aptidão de toda a população (gB).
6. Para cada partícula p em P faça:
    (a) Atualize a velocidade da partícula pela fórmula apresentada anteriormente
    (b) Atualize a posição da particular pela fórmula: pAtual + Velocidade
7. Se condição de término não for alcançada, retorne ao passo 4.

Problema Proposto

Neste relatório o problema que queremos resolver é, minimizar a função *Eggholder,* que é uma função clássica na condução de testes para otimização de funções e é dada por:

$$f(x,\ y)\ =\ -\ (y\ +\ 47)\ sin\sqrt{|x/2\ +\ (y\ +\ 47)|}\ -\ x\ sin\sqrt{|x\ -\ (y\ +\ 47)|}$$

apresentando vários mínimos locais, essa função alcança mínimo global em

$$f(512,\ 404.2319)\ =\ -\ 959.6407$$

E é este valor que gostaríamos que nosso algoritmo encontrasse, ou próximo disso.

Implementação

Antes de adentramos no código implementado, é importante ressaltar que o algoritmo desenvolvido seguiu as especificações abaixo:

• Para limitar a velocidade de uma partícula para que o sistema não extrapole o espaço de busca, são impostos limites (vmax) para seus valores em cada dimensão (d) do espaço de busca:

$$\text{Se } v_i > v_{max} \text{ então } v_i = v_{max},$$
$$\text{Senão se } v_i < -v_{max} \text{ então } v_i = -v_{max}.$$

• No algoritmo, as velocidades iniciais devem ser geradas aleatoriamente, com valores limitados a 15% do espaço de busca de cada parâmetro (~[-77, +77]).
• Assumir como intervalo $x \in [-512,+512]$ e $y \in [-512,+512]$
• Em relação aos valores máximos da partícula, caso este saia do intervalo das variáveis, eles deverão ser corrigidos para o máximo ou mínimo do intervalo [-512, +512], e a velocidade setada em 0.
• O critério de parada será o número de iterações (20, 50 e 100 iterações).
• Considere o tamanho da população como 50 e 100 indivíduos.

Dito isto, podemos iniciar a análise do código gerado, primeiramente, o algortmo foi desenvolvido em python e dividido em quatro arquivos: constants.py, utils.py, Particle.py e main.py

Resumidamente, constants.py possui constantes que definirão o funcionamento do algoritmo, utils.py possui funções genéricas que podem ser usadas por qualquer componente do software, Particle.py é a classe que define uma partícula em nosso campo de estudo, e por fim o arquivo main.py é o responsável por executar o algoritmo do enxame de partículas, vejamos a seguir cada arquivo:

constants.py

```
POPULATION_SIZE = 100  # 50 E 100
X_RANGE = [-512, 512]
Y_RANGE = [-512, 512]
V_RANGE = [X_RANGE[0]*0.15, X_RANGE[1]*0.15]
ITERATIONS = 100  # 20, 50 E 100
C1 = 1
C2 = 1
```

Este arquivo é composto apenas por constantes que irão variar a forma que o algoritmo trabalha, nele, temos as seguintes definições:

- POPULATION_SIZE: Representa a quantidade de partículas que serão criadas;
- X_RANGE: limites para a váriável X;
- Y_RANGE: limites para a váriável Y;
- V_RANGE: Limites para a velocidade da partícula (15% do limites da variável x)

- ITERATIONS: Quantidade de iterações que devem ser feitas;
- C1: Constante 1 ,utilizada para calcular $\phi 1$;
- C1: Constante 2, utilizada para calcular $\phi 2$.

utils.py

```
def setInRange(value, rangeValue):
   if(value < rangeValue[0]):
       return rangeValue[0]
   elif(value > rangeValue[1]):
       return rangeValue[1]
   else:
       return value
```

A função deste arquivo é reunir ferramentas genéricas que podem ser usadas por várias partes do software, neste caso, apenas uma função foi implementada, a **setInRange** esta função recebe dois parâmetros, o *value* e o *rangeValue,* onde range value deve ser um array de duas posições que definem o range onde o *value* deve se enquadrar, caso *value* ultrapasse um dos limites a função retorna o valor do limite ultrapassado, do contrário, retorna o prórpio *value.*

Particle.py

```
import math
import constants
from utils import setInRange
from random import random


class Particle:
   def __init__(self, x, y, velocity):
       self.x = 0
       self.y = 0
       self.velocity = [0, 0]
       self.fitness = 0

       self.setX(x)
       self.setY(y)
       self.setVelocity(velocity)
       self.pBest = [self.x, self.y]
```

```
    def setX(self, x):
        xRange = constants.X_RANGE
        self.x = setInRange(x, xRange)
        if(self.x == xRange[0] or self.x == xRange[1]):
            self.velocity = [0, 0]

    def getX(self):
        return self.x

    def setY(self, y):
        yRange = constants.Y_RANGE
        self.y = setInRange(y, yRange)
        if(self.y == yRange[0] or self.y == yRange[1]):
            self.velocity = [0, 0]

    def getY(self):
        return self.y

    def getPosition(self):
        return [self.getX(), self.getY()]

    def setPosition(self, position):
        self.setX(position[0])
        self.setY(position[1])

    def setVelocity(self, velocity):
        vRange = constants.V_RANGE
        vX = setInRange(velocity[0], vRange)
        vY = setInRange(velocity[1], vRange)
        self.velocity = [vX, vY]

    def getVelocity(self):
        return self.velocity

    def setPBest(self, pBest):
        self.pBest = pBest

    def getPBest(self):
        return self.pBest

    def setFitnessValue(self, fitnessValue):
        self.fitness = fitnessValue
```

```
    def getFitnessValue(self):
        return self.fitness

    def printParticle(self):
        print("Position: [", self.getX(), ",", self.getY(), "]")
        print("Velocity: ", self.getVelocity())
        print("Best Position: [", self.getPBest()
              [0], ",", self.getPBest()[1], "]")
        print("Fitness Value: ", self.getFitnessValue())
```

Este arquivo define uma classe chamada Particle, que tem vários métodos getters e setters que são totalmente opcionais em python mas decidi implementá-los, neste caso, afim de minimizar as ações feitas diretamente pela main, a seguir explicarei as funções mais complexas e essenciais do algoritmo, mas antes, vejamos os atributos desta classe:

```
        self.x = 0
        self.y = 0
        self.velocity = [0, 0]
        self.fitness = 0
```

ela possui os atribuos x e y que em conjunto formam a posição da partícula, o atributo velocity que representa a velocidade da partícula e fitness que representa o quão bem esta particula está em relação ao objetivo.

Vejamos a seguir alguns métodos mais significativos para a classe

setX, setY e setVelocity

Estes métodos setam valores para os atributos x, y e velocity, respectivamente, com a particularidade de respeitarem o range definido para eles, além disso, setX e setY também zeram a velocidade caso os limites sejam desrespeitados.

getPosition, setPosition

Estes métodos trabalham com as variáveis x e y no formato [x, y]

printParticle

Criei também um método que printa as propriedades da particula de forma organizada para facilitar a visualização dos resultados

main.py

```
import math
import constants
from Particle import Particle
import random


def f(x, y):
    return -(y+47) * math.sin(math.sqrt(abs((x/2) + (y+47)))) - x *
math.sin(math.sqrt(abs(x - (y+47))))


def calculateFitnessValue(position):
    x = position[0]
    y = position[1]
    return f(x, y)


def calculateBestPosition(particle):
    actualPosition = particle.getPosition()
    bestPosition = particle.getPBest()

    fitnessActual = calculateFitnessValue(actualPosition)
    bestFitness = calculateFitnessValue(bestPosition)

    if(fitnessActual < bestFitness):
        return actualPosition
    return bestPosition


def getBestParticle(population):
    population.sort(
        key=lambda particle: particle.getFitnessValue())
    return population[0]


def subPositions(posA, posB, c=1):
    a = (posA[0] - posB[0]) * c
```

```
    b = (posA[1] - posB[1]) * c
    return [a, b]


def sumPositions(posA, posB, c=1):
    a = (posA[0] + posB[0]) * c
    b = (posA[1] + posB[1]) * c
    return [a, b]


def updateVelocity(particle, gBest):
    actualPosition = particle.getPosition()
    bestPosition = particle.getPBest()
    bestParticlePosition = gBest.getPosition()

    phi1 = random.uniform(0.0, 1.0) * constants.C1
    phi2 = random.uniform(0.0, 1.0) * constants.C2
    newVelocity = sumPositions(
        subPositions(bestPosition, actualPosition, phi1),
        subPositions(bestParticlePosition, actualPosition, phi2)
    )
    return newVelocity


def updatePosition(particle):
    velocity = particle.getVelocity()
    position = particle.getPosition()
    return sumPositions(position, velocity)


def main():
    # Determine o número de partículas P da população.
    particlesAmount = constants.POPULATION_SIZE

    population = []
    iterations = constants.ITERATIONS
    xRange = constants.X_RANGE
    yRange = constants.Y_RANGE
    vRange = constants.V_RANGE

    # Atribua uma velocidade inicial (v) igual para todas as partículas.
    vX = random.uniform(vRange[0], vRange[1])
    vY = random.uniform(vRange[0], vRange[1])
```

```
initialVelocity = [vX, vY]

# Inicialize aleatoriamente a posição inicial (x) de cada partícula p de P.
for seq in range(particlesAmount):
    x = random.randint(xRange[0], xRange[1])
    y = random.randint(yRange[0], yRange[1])
    particle = Particle(x, y, initialVelocity)
    population.append(particle)

for iteration in range(iterations):
    # Se condição de término não for alcançada
    for particle in population:
        # Calcule sua aptidão fp = f (p).
        fitnessValue = calculateFitnessValue(particle.getPosition())
        particle.setFitnessValue(fitnessValue)

        # Calcule a melhor posição da partícula p até o momento (pB).
        bPosition = calculateBestPosition(particle)
        particle.setPBest(bPosition)

    # Descubra a partícula com a melhor aptidão de toda a população (gB).
    bParticle = getBestParticle(population)

    for particle in population:
        # Atualize a velocidade da partícula
        newVelocity = updateVelocity(particle, bParticle)
        particle.setVelocity(newVelocity)

        # Atualize a posição da particula
        newPosition = updatePosition(particle)
        particle.setPosition(newPosition)

    # print("\n\n\n")
    # print("ITERATION ", iteration)
    # for particle in population:
    #     print("############")
    #     particle.printParticle()

print("Population result: \n")
for particle in population:
    print("############")
    particle.printParticle()
```

```
main()
```

Este arquivo possui várias funções auxiliares que em conjunto com a função main executam o algoritmo do enxame de partículas, a função main executa, basicamente, os passos listados no tópico Enxame de partículas entretanto, para isto, ela utiliza as funções definidas no mesmo arquivo, acima da mesma, que são explicadas a seguir:

f(x, y)

```
def f(x, y):
    return -(y+47) * math.sin(math.sqrt(abs((x/2) + (y+47)))) - x *
math.sin(math.sqrt(abs(x - (y+47))))
```

esta função é pura e simplesmente a implementação da função *Eggholder* em python, implementei-a separado para facilitar futuras manutenções

calculateFitnessValue(position)

```
def calculateFitnessValue(position):
    x = position[0]
    y = position[1]
    return f(x, y)
```

esta função calcula o quão bem posicionada está uma particula, com base em sua posição, por tanto, deve-se passar como parâmetro a posição da partícula no formato [x, y], para o nosso caso, a melhor posição é aquela que possuir menor valor, por tanto o proprio resultado da Eggholder será suficiente.

calculateBestPosition(particle)

```
def calculateBestPosition(particle):
    actualPosition = particle.getPosition()
    bestPosition = particle.getPBest()

    fitnessActual = calculateFitnessValue(actualPosition)
    bestFitness = calculateFitnessValue(bestPosition)
```

```
    if(fitnessActual < bestFitness):
        return actualPosition
    return bestPosition
```

Como vimos anteriormente, guardar a melhor posição da partícula é uma etapa importante do algoritmo, e esta é nesta tarefa que esta função auxilia, ela calcula se a melhor posição da particula é a que ela está agora ou ainda se mantem aquela setada como melhor posição anteriormente, a escolha é feita com base no calculo de fitness de cada posição.

getBestParticle(population)

```
def getBestParticle(population):
    population.sort(
        key=lambda particle: particle.getFitnessValue())
    return population[0]
```

Esta função é responsavel por encontrar o melhor indivíduo da população, ou enxame, ela realiza esta tarefa da seguinte forma: ordena a lista de partículas de acordo com seu *fitnessValue* do menor para o maior e retorna o primeiro individuo da lista, já que este tem o menor valor, que é o que buscamos.

subPositions e sumPositions

```
def subPositions(posA, posB, c=1):
    a = (posA[0] - posB[0]) * c
    b = (posA[1] - posB[1]) * c
    return [a, b]


def sumPositions(posA, posB, c=1):
    a = (posA[0] + posB[0]) * c
    b = (posA[1] + posB[1]) * c
    return [a, b]
```

estas duas funções auxiliam na manipulação das posições da partícula, subtraindo ou somando valores a elas, sempre no formato [x, y], além disso, elas adicionam a opção de multiplicar uma constante aos valores resultantes, que, por padrão, é configurado para 1, afim de não alterar os resultados

updateVelocity(particle, gBest)

```
def updateVelocity(particle, gBest):
    actualPosition = particle.getPosition()
    bestPosition = particle.getPBest()
    bestParticlePosition = gBest.getPosition()

    phi1 = random.uniform(0.0, 1.0) * constants.C1
    phi2 = random.uniform(0.0, 1.0) * constants.C2
    newVelocity = sumPositions(
        subPositions(bestPosition, actualPosition, phi1),
        subPositions(bestParticlePosition, actualPosition, phi2)
    )
    return newVelocity
```

Aqui calculamos a nova velocidade de uma partícula seguindo a formula da velocidade apresentada anteriormente, para isso recebemos os parametros *particle* que é a particula a ser calculada e *gBest* que é a melhor particula do enxame, com essas informações, calculamos phi1 ($\phi 1$) e phi2 ($\phi 2$) e calculamos a nova velocidade com o auxilio das funções sumPositions e subPositions apresentadas anteriormente.

updatePosition(particle)

```
def updatePosition(particle):
    velocity = particle.getVelocity()
    position = particle.getPosition()
    return sumPositions(position, velocity)
```

Por fim, podemos gerar a nova posição de uma particula através da updatePosition, que recebe a particula como parâmetro e soma sua posição com sua velocidade.

Resultados

A seguir serão apresentados os resultados obtidos com o código implementado anteriormente, primeiro será apresentado uma análise estatística dos resultados obtidos para 10 execuções, descrevendo numa tabela o melhor, a média dos resultados e o desvio padrão, em seguida apresentaremos o melhor resultado e média para cada numero de iterações (20, 50 e 100) em formato de gráfico e por fim uma análise dos resultados e eficácia do algoritmo sera apresentada como conclusão do relatório.

Análise Estatística

Para esta análise, executamos o algoritmo dez vezes com um enxame de tamanho igual a 50 particulas e cada execução sofreu 50 iterações, coletamos os valores de melhor resultado, resultado médio e desvio padrão, que são apresentados abaixo em forma de tabela

Melhor Resultado	Média	Desvio Padrão
-715,3103996	-126,207955	462,3879893
-651,9737745	-208,5948263	469,2238042
-565,997243	-413,6617019	266,1866258
-893,2756921	-425,3657307	607,7603345
-716,0986465	-367,2225137	451,0250821
-895,4246081	-363,2262597	549,3068336
-701,1238352	-192,3645632	486,5849372
-890,3518069	-97,75202013	562,3244179
-766,7119462	-271,8126092	522,9258134
-625,0040116	-370,957511	371,8659704

Tabela 1. Analise estatistica de 10 execuções do algoritmo

Como visto acima, apesar de termos conseguido bons resultados, como -895,4246081, nenhuma partícula conseguiu alcançar o mínimo global que resultaria em ≈ -959.6407. Além disso vimos que a média ficou bem distante do valor desejado, tivemos também um desvio padrão muito alto, o que indica uma alta dispersão. Com base nisso, podemos concluir que o melhor a se fazer é descartar alguns resultados, levando em consideração apenas o melhores, afim de diminuir a dispersão.

Gráficos

Abaixo serão apresentados os graficos dos resultados obtidos (melhor resultado e média) para cada número de iterações (20, 50 e 100). Manteremos a quantidade de 50 particulas por enxame.

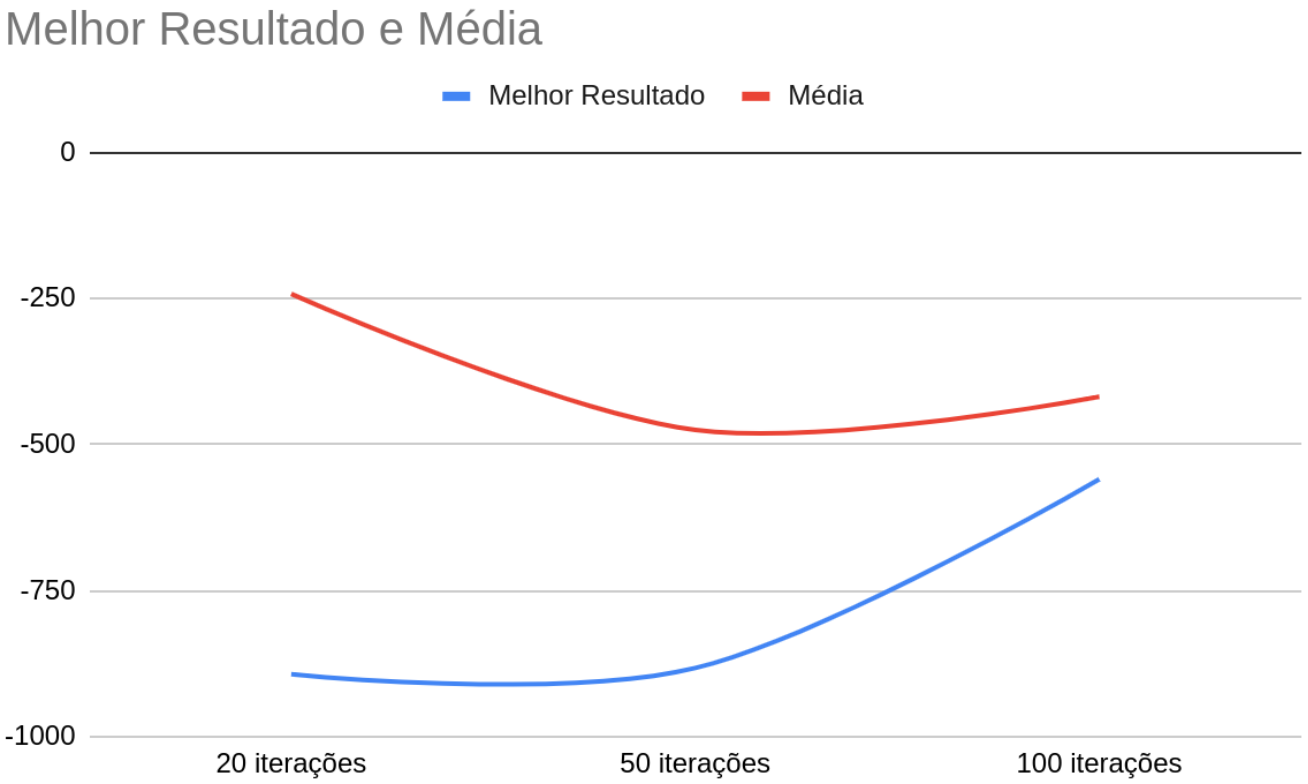

Grafico 1. Melhor resultado e media em relação as iterações

É importante lembrar que para a análise deste gráfico, quanto menor o valor, melhor, ou seja, quanto mais próximo de -1000 na escala do gráfico, mais nos aproximamos do mínimo da função, tendo isto em vista podemos ver que a quantidade de iterações não interferiu muito nos resultados obtidos, com 100 iterações tivemos o pior resultado, entretanto, pudemos observar que quanto mais iterações realizavamos mais a média se aproximava do melhor resultado, ou seja, quanto mais iterações mais as particulas convergem para o melhor resultado, elas se aproximam, outra prova disto se dá quando analisamos as 20 iterações, neste caso, conseguimos um resultado muito bom, porém a média ficou muito distante do desejado.

Conclusão

Após a analise dos resultados obtidos, podemos concluir que o algoritmo de enxame de partículas é um algoritmo relativamente simples de ser implementado e muito útil para alguns problemas de otimização, entretanto é importante entender suas limitações e comportamentos, como pudemos ver, raramente o algoritmo nos leva ao resultado ótimo, onde não há mais como melhorar, mas traz bons resultados com um esforço computacional relativamente baixo, por tanto pode ser uma boa opção quando não for necessário se obter altas precições na acurácia dos resultados.