Vitor Galioti Martini

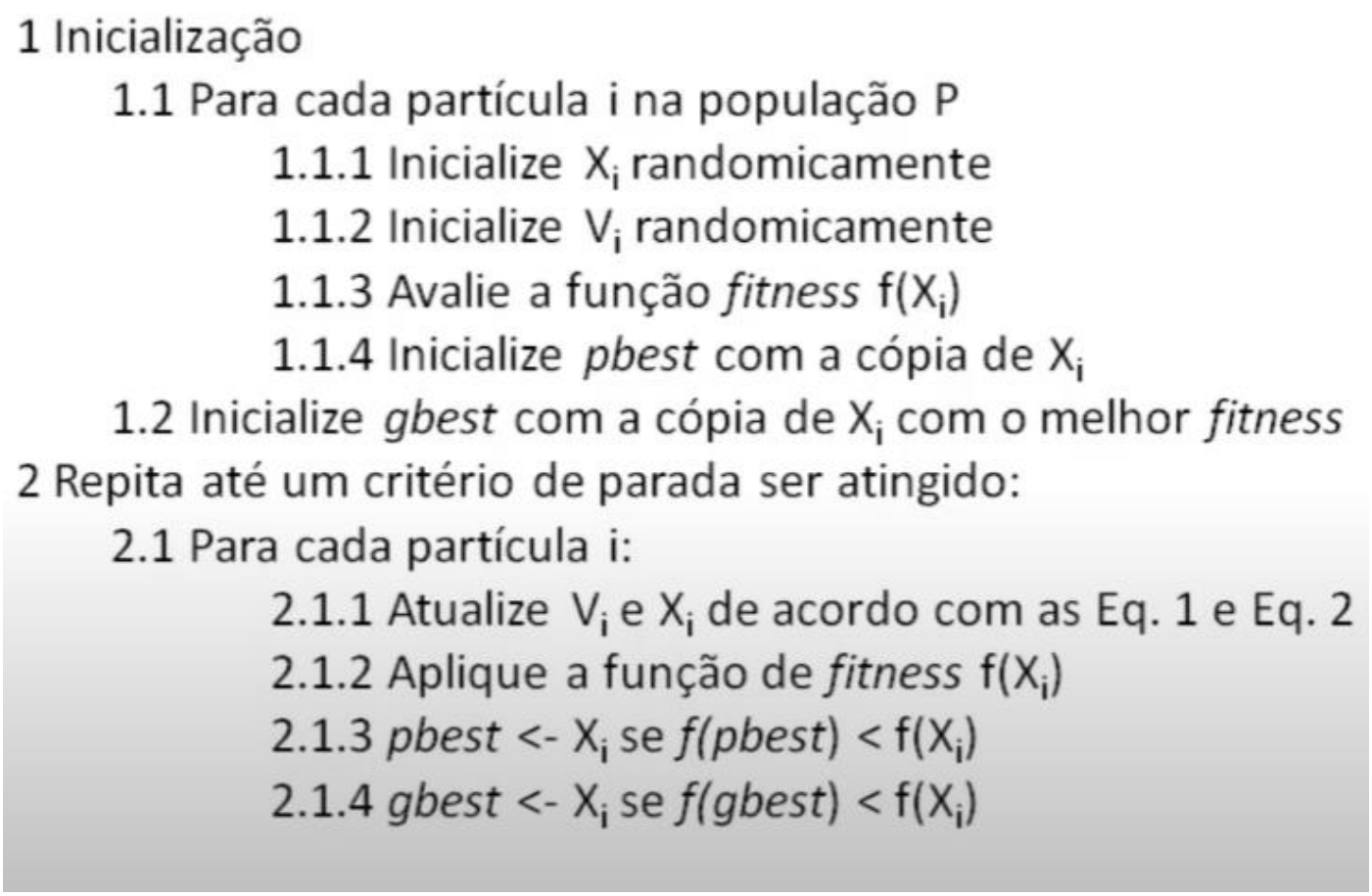
135543

Aula 07 - Exercício prático algoritmos bioinspirados

Abaixo meu algoritmo PSO e sua análise descrita em comentários:

```
import random

# Função f, recebe a equação do usuário e a resolve com a função eval
def f(funcao, x):
    return eval(funcao)

# Algoritmo PSO
def PSO():
    # Recebe as entradas do usuário:
    funcao = input('f(x) = ')
    w = float(input('w = '))
    c1 = float(input('c1 = '))
    c2 = float(input('c2 = '))
    r1 = float(input('r1 = '))
    r2 = float(input('r2 = '))
    n = int(input('n = '))
    iteracoes = int(input('iteracoes = '))

    # Gera partículas para X e V randomicamente
    x = []
    v = []
    for i in range(n):
        x.append(float("{:.4f}".format((random.uniform(0, 1) - 0.5) * 10)))
        v.append(float("{:.4f}".format(random.uniform(0, 1) - 0.5)))

    # Entradas usadas no exercício teórico:
    # x = [-0.343, 3.956, -1.123, -0.098, 0.039]
    # v = [0.0319, 0.3185, 0.3331, 0.2677, -0.3292]

    # Inicia PBest como uma cópia de X e Gbest como a primeira partícula de X
    Pbest = x
    Gbest = x[0]
    Fbest = float("{:.4f}".format(f(funcao, 0)))

    # Percorre as partículas de X comparando cada uma para chegar no melhor valor de Gbest
    for i in x:
        if f(funcao, i) > Gbest:
            Gbest = i
            Fbest = float("{:.4f}".format(f(funcao, i)))

    # Printa os valores
    print("Local best position: ", Pbest)
    print("Global best fitness: ", Fbest)
    print("Global best position: ", Gbest)
    print(" ")

    # A execução acima foi a primeira interação, agora executa as restantes:
    while iteracoes - 1 > 0:

        # Percorre partículas
        for i in range(n):
            # Define os valores V e X conforme equações vistas em aula
            v[i] = float("{:.4f}".format(w * v[i] + c1 * r1 * (Pbest[i] - x[i]) + c2 * r2 * (Gbest - x[i])))
            x[i] = float("{:.4f}".format(x[i] + v[i]))
            # Redefine PBest caso encontre um valor que o maximize
            if f(funcao, Pbest[i]) < f(funcao, x[i]):
                Pbest = x[i]
            # O mesmo para Gbest
            if f(funcao, Gbest) < f(funcao, x[i]):
                Gbest = x[i]
                Fbest = float("{:.4f}".format(f(funcao, x[i])))

        # Printa os valores
        print("Local best position: ", Pbest)
        print("Global best fitness: ", Fbest)
        print("Global best position: ", Gbest)
        print(" ")

        iteracoes -= 1

# Executa o algoritmo
PSO()
```

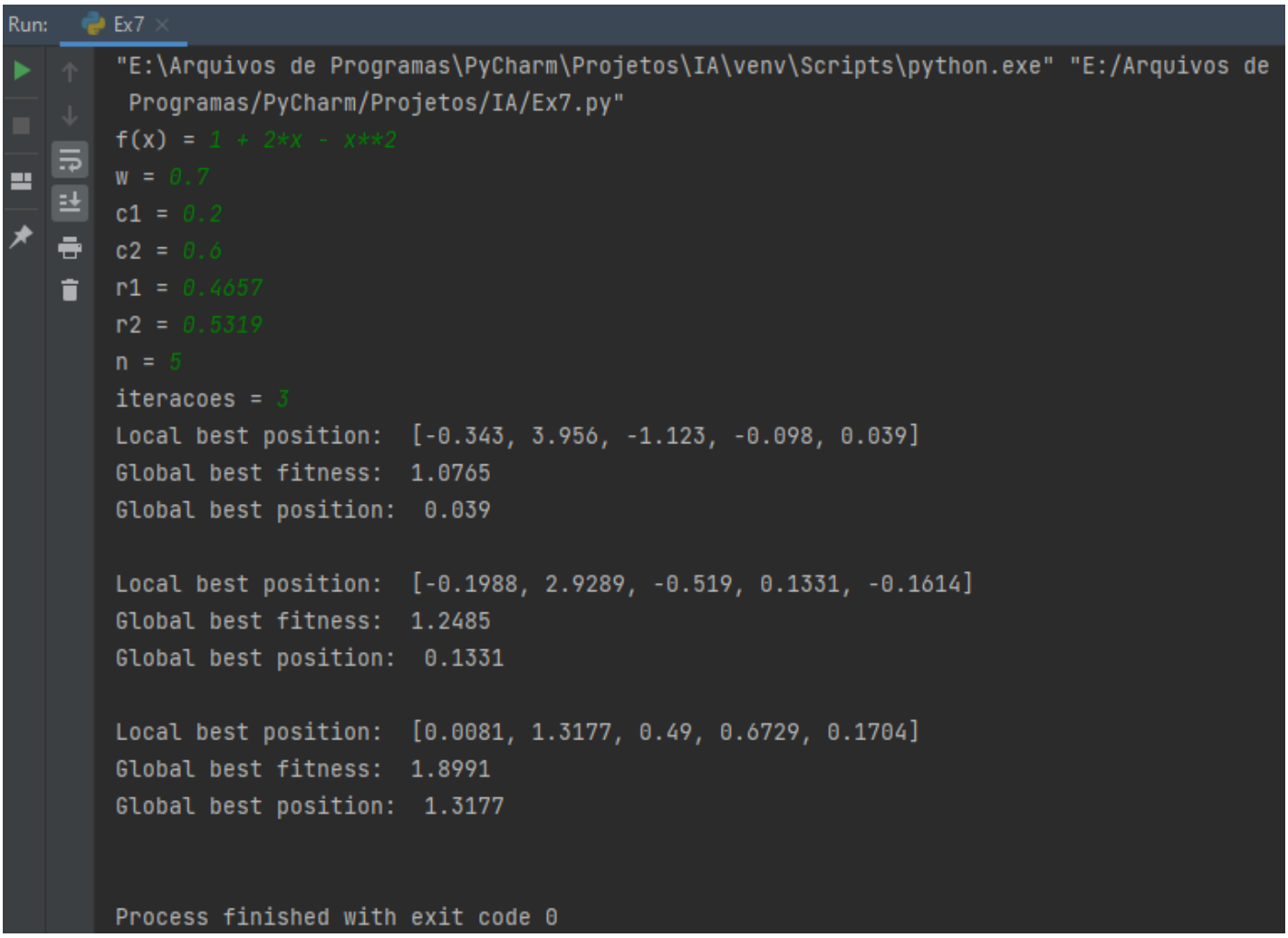
Minha ideia para a implementação do algoritmo PSO foi seguir o pseudocódigo visto em aula:



Para gerar a saída do exercício teórico proposto descomentei a inserção de valores manual:

```
# Gera partículas para X e V randomicamente
x = []
v = []
for i in range(n):
    x.append(float("{:.4f}".format((random.uniform(0, 1) - 0.5) * 10)))
    v.append(float("{:.4f}".format(random.uniform(0, 1) - 0.5)))

# Entradas usadas no exercício teórico:
# x = [-0.343, 3.956, -1.123, -0.098, 0.039]
# v = [0.0319, 0.3185, 0.3331, 0.2677, -0.3292]
```

A saída encontrada foi a seguinte:



Abaixo meu algoritmo ACO e sua análise descrita em comentários:

```
# Algoritmo ACO
class Grafo:

    # Construtor do grafo
    def   init  (self, vertices, alpha, beta, rho, tau):
        self.vertices = vertices
        self.caminho = [[] for i in range(self.vertices)]
        self.feromonio = [[] for i in range(self.vertices)]
        self.visitados = [0 for i in range(vertices)]
        self.a = alpha
        self.b = beta
        self.r = rho
        self.t = tau

    #Percorre cada aresta e recebe o valor do usuário
    def adicionar valor aresta(self):
        for i in range(self.vertices):
            if i != 0:
                print("Vértice ", i, ": ")
                for j in range(self.vertices):
                    if j != 0:
                        if j != i:
                            mensagem = "Aresta (" + str(i) + " " + str(j) + "), peso: "
                            valor = int(input(mensagem))
                            self.caminho[i].append([j, valor])
                            self.feromonio[i].append([j, self.t])
                        else:
                            self.caminho[i].append([j, 0])
                            self.feromonio[i].append([j, 0])

    # Calcula a probabilidade daquele caminho ser tomado
    def probabilidade(self, vertice atual, i):
        normalizacao = 0
        for j in range(self.vertices - 1):
            if self.visitados[self.caminho[vertice_atual][j][0]] == 0 or j == vertice_atual:
                normalizacao += (self.feromonio[vertice_atual][j][1] * self.a * self.b * 1 /
self.caminho[vertice_atual][j][1])

        return float("{:.2f}".format(((self.feromonio[vertice atual][i - 1][1] * self.a * self.b * 1 /
self.caminho[vertice_atual][i - 1][1]) / normalizacao)))

    # Caminhar
    def caminhar(self, comeco):
        # Começa do vértice atual
        percurso = [comeco]
        vertice_atual = comeco
        i = 0

        # Seta os visitados para zero
        self.visitados = [0 for i in range(self.vertices)]

        # Enquanto houver vértices
        while i < self.vertices - 1:

            # Reseta variável que recebe o melhor vértice
            Gbest = None

            # Caso o vértice ainda não foi visitado
            if self.visitados[vertice atual] == 0:
                self.visitados[vertice atual] = 1

                # Percorre os vértices adjacentes
                for vertices in self.caminho[vertice_atual]:
                    # Caso não seja o vértice atual (pois o vértice A se liga com ele mesmo com peso 0)
                    if self.visitados[vertices[0]] == 0:
                        # Se ainda não tem melhor vértice definido, seta o primeiro encontrado
                        if Gbest is None:
                            Gbest = vertices[0]
                            Fbest = self.probabilidade(vertice_atual, Gbest)
                        else: # Caso contrário, verifica a probabilidade do vértice analisado ser o melhor,
se for, armazena em Gbest
                            if self.probabilidade(vertice atual, vertices[0]) > Fbest:
                                Gbest = vertices[0]
                                Fbest = self.probabilidade(vertice_atual, vertices[0])

            # Caso não tenha chegado no fim, adiciona o vértice no percurso
            if Gbest is not None:
                vertice atual = Gbest
                percurso.append(Gbest)

            i += 1
```

```
        print("")
        print("Percurso: ", percurso)

    # Atuliza o valor do feromônios
    def atualizar_feromonio(self):
        # Percorre cada feromônio e atualiza seu valor com a formula
        for i in range(self.vertices):
            if i != 0:
                for j in range(self.vertices - 1):
                        if self.caminho[i][j][1] != 0:
                            self.feromonio[i][j][1] = float("{:.2f}".format((1 -
self.r)*self.feromonio[i][j][1]+self.r*1/self.caminho[i][j][1]))

    # Exibe o valor dos feromônios
    def exibir feromonio(self):
        print("")
        print("Feromônio:")
        print('  ', end=' ')
        for i in range(self.vertices):
            if i != 0:
                if i == self.vertices - 1:
                    print(i)
                else:
                    print(i, end='   ')

        for i in range(self.vertices):
            if i != 0:
                print(i, end='  ')
                for j in self.feromonio[i]:
                    print(j[1], end='   ')

                print("")

    # Exibe a qualidade das arestas
    def exibir qualidade aresta(self):
        print("")
        print("Qualidade das arestas")
        print('  ', end=' ')

        for i in range(self.vertices):
            if i != 0:
                if i == self.vertices - 1:
                    print(i)
                else:
                    print(i, end='   ')

        for i in range(self.vertices):
            if i != 0:
                print(i, end='  ')
                for j in self.caminho[i]:
                    print(j[1], end='   ')

                print("")

        print("")

    # Executa o algoritmo começando de cada vertice
    def executar(self):
        i = 1
        # Enquanto houver vértices
        while i < self.vertices:

            # Caminha para os vértices vizinhos
            self.caminhar(i)

            # Atualiza os feromônios
            self.atualizar_feromonio()

            # Mostra o feromônio
            self.exibir feromonio()

            i += 1

def ACO():
    # Recebe as entradas do usuário
    qtd vertices = int(input("Informe o n° de vértices: "))
    a = float(input("Informe alpha: "))
    b = float(input("Informe beta: "))
    r = float(input("Informe rho: "))
    t = float(input("Informe tau: "))

    # Monta o grafo
    g = Grafo(qtd_vertices + 1, a, b, r, t)
```

```
    # Adiciona as arestas
    g.adicionar_valor_aresta()

    # Exibe a qualidade das arestas
    g.exibir_qualidade_aresta()

    # Começa o percurso
    g.executar()

# Executa o algoritmo
ACO()
```

Sua estrutura foi baseada nos passos vistos em aula:

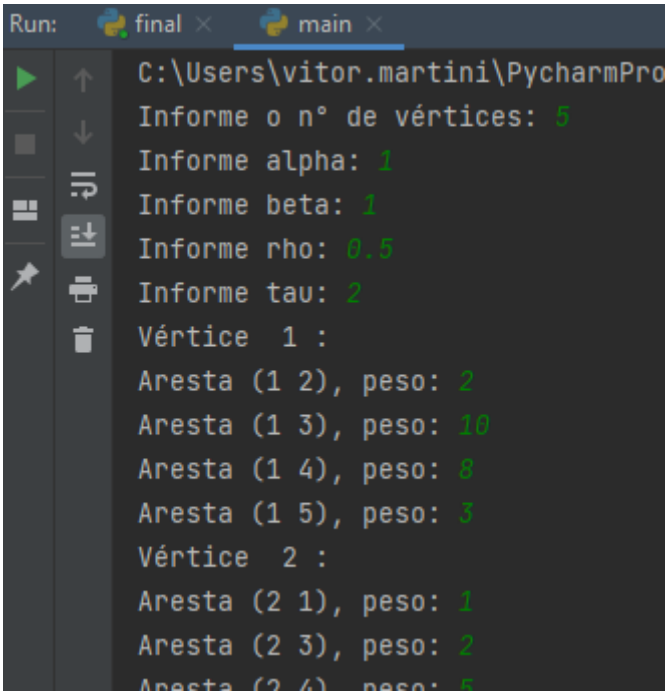
Passos do algoritmo de colônias de formigas (ACO)

1. **Inicialização**: cada formiga inicia em um estado.

2. **Construção de soluções**: as formigas artificiais se movem através de estados adjacentes de um problema de acordo com uma transição, criando iterativamente soluções.

3. **Atualização de feromônio**: executa atualizações na trilha de feromônio.
   - 2.1 Reforço de trilhas de feromônios: atualizar sempre que uma boa solução foi criada ou atualizadas após cada iteração.
   - 2.2 Evaporação de trilha de feromônio: ajuda as formigas a esquecer as más soluções que foram aprendidas no início da execução do algoritmo.

Replicando as entradas do exercício visto em aula no código desenvolvido:



Temos a saída:

```
Qualidade das arestas
   1   2   3   4   5
1  0   2   10   8   3
2  1   0   2   5   7
3  9   1   0   3   6
4  10   4   3   0   2
5  2   7   5   1   0


Percurso:  [1, 2, 3, 4, 5]

Feromônio:
   1   2   3   4   5
1  0   1.25   1.05   1.06   1.17
2  1.5   0   1.25   1.1   1.07
3  1.06   1.5   0   1.17   1.08
4  1.05   1.12   1.17   0   1.25
5  1.25   1.07   1.1   1.5   0
```

```
Percurso:  [2, 1, 5, 4, 3]

Feromônio:
   1   2   3   4   5
1  0   0.88   0.58   0.59   0.75
2  1.25   0   0.88   0.65   0.61
3  0.59   1.25   0   0.75   0.62
4  0.58   0.69   0.75   0   0.88
5  0.88   0.61   0.65   1.25   0

Percurso:  [3, 2, 1, 5, 4]

Feromônio:
   1   2   3   4   5
1  0   0.69   0.34   0.36   0.54
2  1.12   0   0.69   0.43   0.38
3  0.35   1.12   0   0.54   0.39
4  0.34   0.47   0.54   0   0.69
5  0.69   0.38   0.43   1.12   0
```

```
Percurso:  [4, 5, 1, 2, 3]

Feromônio:
   1   2   3   4   5
1  0   0.59   0.22   0.24   0.44
2  1.06   0   0.59   0.32   0.26
3  0.23   1.06   0   0.44   0.28
4  0.22   0.36   0.44   0   0.59
5  0.59   0.26   0.32   1.06   0

Percurso:  [5, 4, 3, 2, 1]

Feromônio:
   1   2   3   4   5
1  0   0.54   0.16   0.18   0.39
2  1.03   0   0.54   0.26   0.2
3  0.17   1.03   0   0.39   0.22
4  0.16   0.3   0.39   0   0.54
5  0.54   0.2   0.26   1.03   0

Process finished with exit code 0
```

Percurso, apresentado por 1, 2, 3, 4, 5 se refere aos vértices A, B, C, D, E. Então “Percurso: [1, 2, 3, 4, 5]” se refere a fazer o caminho A -> B ->C ->D ->E.

O código implementado começa do primeiro vértice, e percorre os seguintes. Ao finalizar o percurso, atualiza a tabela de feromônios e percorre os vértices novamente, porém partindo de um vértice diferente do anterior. O processo se repete até que já tenham sido percorridos os caminhos começando de cada vértice.