IA – 1° SEMESTRE DE 2022

7. EXERCÍCIO PRÁTICO – ALGORITMOS BIOINSPIRADOS

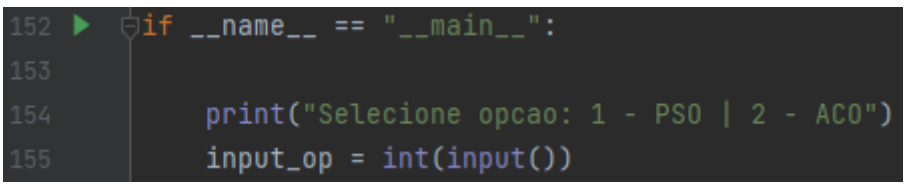
Nome: Luiz Gustavo Alves Assis da Silva
RA: 149115

1) Implemente o PSO e ACO, considerando como entrada a leitura de qualquer função/grafo e parâmetros. Confirme a resposta dos exercícios anteriores com sua implementação anexando um *print* da saída.

```
'''
#
#   AULA 7
#   ALGORITMOS BIOINSPIRADOS (PSO - ACO)
#   LINGUAGEM DE PROGRAMAÇÃO: Python
#
#   NOME: LUIZ GUSTAVO ALVES ASSIS DA SILVA
#   RA: 149115
#

'''

# BIBLIOTECAS USADAS:
import math

class Grafo:
    def __init__(self, vertices, arestas, pesos):
        self.vertices = vertices
        self.arestas = arestas
        self.pesos = pesos
```

Temos um programa que implementa os seguintes algoritmos bioinspirados:
PSO (Particle Swarm Optimization – Otimização de Enxame de Partículas).
ACO (Ant Colony Optimization – Otimização de Colônia de Formigas).

O objetivo deste programa é receber, como entrada, a leitura de qualquer função/grafo e parâmetros de acordo com a escolha de algoritmo e imprimir a solução do problema como saída.



```
if __name__ == "__main__":

    print("Selecione opcao: 1 - PSO | 2 - ACO")
    input_op = int(input())
```

Primeiramente, na função principal, foi feito um menu no terminal para auxiliar o usuário a escolher os parâmetros dos algoritmos de acordo com a escolha inicial de opção, sendo a opção 1 – Algoritmo PSO e opção 2- Algoritmo ACO.

ALGORITMO PSO:

```
157     if (input_op == 1): # EXECUTAR PSO
158
159         print("Insere os seguintes parâmetros: iteracoes | w | c1 | c2 | r1 | r2")
160         iteracoes = int(input())
161         w = float(input())
162         c1 = float(input())
163         c2 = float(input())
164         r1 = float(input())
165         r2 = float(input())
166
167         print("Insere os seguintes parâmetros: x | y")
168         X = list(map(float, input().split()))
169         Y = list(map(float, input().split()))
170
171         print("Insere os seguintes parâmetros: f")
172         f = input()
```

Caso a opção do usuário seja a de utilizar o algoritmo PSO, é preciso que o mesmo especifique os parâmetros que serão utilizados no algoritmo - a exemplo do número total de iterações, vetor posição x, vetor velocidade y, a função *f*, etc.

```
174         Pbest, Gbest = eq2(X, f)
175         print("Iteração  1  - Pbest: {:.5}".format(Pbest), " - Gbest: {:.5}".format(Gbest))
176
177         for i in range(1, iteracoes):
178             newPbest, newGbest = eq1(X, Y, w, c1, c2, r1, r2, f, Pbest, Gbest)
179
180             Pbest = newPbest
181             Gbest = newGbest
182
183             print("Iteração ", i + 1, " - Pbest: {:.5}".format(Pbest), " - Gbest: {:.5}".format(Gbest))
```

Em seguida, é realizada a primeira chamada na função eq2 (será vista adiante), passando como parâmetro o vetor posição x e a função *f*, retornando a melhor posição alcançada por si (Pbest) e a melhor posição alcançada pelos vizinhos (Gbest). Após isso, dentro de um loop é feito aplicação do Algoritmo PSO, passando todos os argumentos dados como parâmetro na função eq1 e atualizando os valores de Pbest e Gbest. Vale ressaltar que a condição de parada do loop é dada quando for completado o número total de iterações.

```python
def eq2(X, f):
    temp = -1
    for i in range(len(X)):
        x = X[i]
        Pbest = eval(f)
        if (Pbest > temp):
            temp = Pbest
            Gbest = X[i]

    Pbest = temp
    return Pbest, Gbest

def eq1(X, Y, w, c1, c2, r1, r2, f, Pbest, Gbest):

    V = []
    for i in range(len(Y)):
        v = w * Y[i] + (c1 * r1) * (Pbest - X[i]) + (c2 * r2) * (Gbest - X[i])
        V.append(X[i] + v)

    return eq2(V, f)
```

Para a função eq2, é atualizado as posições das partículas, por meio da função inserida pelo usuário, e as escolhas de Pbest e Gbest, melhor posição alcançada por si e melhor posição alcançada pelos vizinhos, respectivamente.

Para a função eq1, é denotado os parâmetros para o movimento das partículas e, após a definição, é feita uma chamada de função para eq2 passando o vetor movimento e a função como parâmetro.

```
Selecione opcao: 1 - PSO | 2 - ACO
1
Insere os seguintes parâmetros: iteracoes | w | c1 | c2 | r1 | r2
3
0.70
0.20
0.60
0.4657
0.5319
Insere os seguintes parâmetros: x | y
-0.343 3.956 -1.123 -0.098 0.039
0.0319 0.3185 0.3331 0.2677 -0.3292
Insere os seguintes parâmetros: f
1+2*x-x**2
Iteração  1  - Pbest: 1.0765  - Gbest: 0.039
Iteração  2  - Pbest: 1.4262  - Gbest: 0.2425
Iteração  3  - Pbest: 1.5644  - Gbest: 0.34002

Process finished with exit code 0
```

(Exemplo de entrada e saída do algoritmo PSO)

ALGORITMO ACO:

```
185     elif(input_op == 2): # EXECUTAR ACO
186         print("Insere os seguintes parâmetros: alfa | beta | tau | ro")
187         alfa = float(input())
188         beta = float(input())
189         tau = float(input())
190         ro = float(input())
191 
192         print("Insere os seguintes parâmetros: vertices | pesos | fermonio")
193         v = list((input().split()))
194         w_not_ord = list(map(int, input().split()))
195         frm_not_ord = list(map(int, input().split()))
```

Caso a opção do usuário seja a de utilizar o algoritmo ACO, é preciso que o mesmo especifique os parâmetros que serão utilizados no algoritmo - a exemplo dos vertices e pesos do grafo de qualidade de arestas, a matriz de feromonio, etc.

```
197             # FORMATANDO GRAFO E MATRIZ FEROMONIO
198             e = list()
199             w = list()
200             frm = list()
201 
202             k = 0
203             for i in range(len(v)):
204                 elem_w = []
205                 elem_frm = []
206                 for j in range(len(v)):
207 
208                     if i != j:
209                         e.append((v[i], v[j]))
210 
211                     elem_w.append(w_not_ord[k])
212                     elem_frm.append(frm_not_ord[k])
213                     k += 1
214 
215                 w.append(elem_w)
216                 frm.append(elem_frm)
217 
218             g = Grafo(v, e, w)
219             matriz = g.obterMatrizDeAdjacencia()
```

Antes de prosseguir com o algoritmo ACO, é preciso formatar as entradas das matrizes de peso e feromônio, além da criação das arestas "*e* (edges)" para facilitar o desenvolvimento do código das funções auxiliares. Por fim, nosso grafo é criado por meio do construtor da classe Grafo passando os vértices, arestas e pesos como parâmetros do método.

```
221         print("Insere os seguintes parâmetros: vertice inicial")
222         vertice = input()
223 
224         solucao = g.ACO(matriz, vertice, frm, alfa, beta, tau, ro)
225         print("Caminho da Solucao: ", solucao)
226         print("Feromonio: ")
227         for i in range(len(frm)):
228             print(frm[i])
```

A seguir, é requisitado o último parâmetro do algoritmo, sendo ele o vértice inicial para realizar a busca do algoritmo ACO. Por último, é feita a chamada na função ACO passando todos os argumentos como parâmetro, retornando a solução do problema (caminho) e o ferômonio atualizado.

```
22      def converterStringInt(self):
23          v_list = []
24          index = []
25 
26          for i in self.vertices:
27              v_list.append(i)
28              index.append(v_list.index(i))
29 
30          return index, v_list
31 
32      def mapearStringInt(self, index, v_list, vertice):
33          i = 0;
34          for u in v_list:
35              if (u == vertice):
36                  return index[i]
37              i += 1
```

As funções auxiliares acima permitem a manipulação do grafo com maior facilidade, efeutando a conversão de uma string para inteiro além de mapear as posições dos índices na matriz de adjacência.

```
39  def obterMatrizDeAdjacencia(self):
40      index, v_list = self.converterStringInt()
41      numeroDeVertices = len(self.vertices)
42 
43      matriz = [[0 for col in range(numeroDeVertices)] for row in range(numeroDeVertices)]
44 
45      k = 0
46      for i in self.vertices:
47          m = 0
48          for j in self.vertices:
49              u = self.mapearStringInt(index, v_list, i)
50              v = self.mapearStringInt(index, v_list, j)
51              matriz[u][v] = self.pesos[k][m]
52              m += 1
53          k += 1
54 
55      return matriz
```

A função obterMatrizDeAdjacencia, como o nome diz, converte o grafo inserido do usuário para uma matriz de adjacencia, permitindo o acesso em tempo constante de um peso no grafo de qualidade de aresta.

```python
def obterAdjacentesNaMatrizDeAdjacencia(self, matriz, vertice, vertice_visitado):
    Adjacentes = []
    index, v_list = self.converterStringInt()
    index_vertice = self.mapearStringInt(index, v_list, vertice)

    for i in range(len(matriz)):

        if matriz[index_vertice][i] != 0:
            j = self.mapearStringInt(v_list, index, i)
            exist_count = vertice_visitado.count(j)

            if exist_count == 0:
                Adjacentes.append(j)

    return Adjacentes
```

A função obterAdjacentesNaMatrizDeAdjacencia, como o nome diz, retorna a lista de vértices adjacentes (vizinhos) para um dado vértice específicado pelo parâmetro da função. Cabe ressaltar que foi feita uma modificação no código para que a lista de adjacentes não inclua vértices que já foram visitados durante as iterações do algoritmo.

```python
def somatorio(self, Adjacentes, vertice, alfa, beta, tau):
    index, v_list = self.converterStringInt()
    pos = self.mapearStringInt(index, v_list, vertice)
    sum = 0

    for i in Adjacentes:
        j = self.mapearStringInt(index, v_list, i)
        eta = self.pesos[pos][j]
        sum += ((alfa * tau) * (beta * (1 / eta)))

    return sum, pos;

def calcProbabilidade(self, Adjacentes, alfa, beta, tau, sum, pos):
    index, v_list = self.converterStringInt()
    temp = - 1

    for j in Adjacentes:
        k = self.mapearStringInt(index, v_list, j)
        eta = self.pesos[pos][k]

        P = ((alfa * tau) * (beta * (1 / eta))) / sum
        if (P > temp):
            temp = P
            vertice = j

    return vertice
```

As funções auxiliares acima realizam a construção da solução do algoritmo ACO – calculando o somatório da quantidade de feromônio e qualidade da aresta, calculando probabilidade da formiga optar pela aresta (i, j).

```
100     def atualizarFeromonio(self, matriz, vertice, feromonio, Adjacentes, ro):
101         index, v_list = self.converterStringInt()
102         i = self.mapearStringInt(index, v_list, vertice)
103 
104         for j in range(len(feromonio)):
105             if (i != j):
106 
107                 k = self.mapearStringInt(v_list, index, j)
108                 exist_count = Adjacentes.count(k)
109                 if exist_count != 0:
110                     feromonio[i][j] = ((1 - ro) * feromonio[i][j]) + ro * (1 / matriz[i][j])
```

Por fim, a útlimas função auxiliar acima atualiza o feromônio de todas arestas para um vertice dado pelo parâmetro da função.

```
112     def ACO(self, matriz, vertice, feromonio, alfa, beta, tau, ro):
113         index, v_list = self.converterStringInt()
114         vertice_visitado = list()
115 
116         solucao = list()
117         solucao.append(vertice)
118 
119         for i in range(len(matriz) - 1):
120             Adjacentes = self.obterAdjacentesNaMatrizDeAdjacencia(matriz, vertice, vertice_visitado)
121             vertice_visitado.append(vertice)
122             self.atualizarFeromonio(matriz, vertice, feromonio, Adjacentes, ro)
123 
124             sum, pos = self.somatorio(Adjacentes, vertice, alfa, beta, tau)
125             vertice = self.calcProbabilidade(Adjacentes, alfa, beta, tau, sum, pos)
126             solucao.append(vertice)
127 
128         return solucao
```

A função ACO realiza a implementação do algoritmo ACO, em que uma lista de vértices visitados serve como "flag" para indicar vértices já visitados e uma lista solução é criada para armazenar os vértices percorridos pela formiga. Dentro do loop, é feita a chamada da função para obter os adjacentes do vértices vizinhos e a lista de vértices visitados armazena esse vértice, em seguida, o feromônio é atualizado para este vértice.

Por fim, é feito o cálculo da somatória e cálculo das probabilidades retornando o vértice com maior probabilidade de ser escolhido. Sendo assim, o vertice é atualizado e a lista solução armazena este vertice.

```
Selecione opcao: 1 - PSO | 2 - ACO
2
Insere os seguintes parâmetros: alfa | beta | tau | ro
1
1
2
0.5
Insere os seguintes parâmetros: vertices | pesos | fermonio
A B C D E
0 2 10 8 3 1 0 2 5 7 9 1 0 3 6 10 4 3 0 2 2 7 5 1 0
0 2 2 2 2 2 0 2 2 2 2 2 0 2 2 2 2 2 0 2 2 2 2 2 0
Insere os seguintes parâmetros: vertice inicial
A
Caminho da Solucao:  ['A', 'B', 'C', 'D', 'E']
Feromonio:
[0, 1.25, 1.05, 1.0625, 1.1666666666666667]
[2, 0, 1.25, 1.1, 1.0714285714285714]
[2, 2, 0, 1.1666666666666667, 1.0833333333333333]
[2, 2, 2, 0, 1.25]
[2, 2, 2, 2, 0]

Process finished with exit code 0
```

(Exemplo de entrada e saída do algoritmo ACO)