



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



درس حسابگری زیستی

تمرین شماره 3

نام و نام خانوادگی : پرهام زیلوچیان مقدم

شماره دانشجویی : 810198304

خرداد 1400

سوال اول: 3

سوال اول :

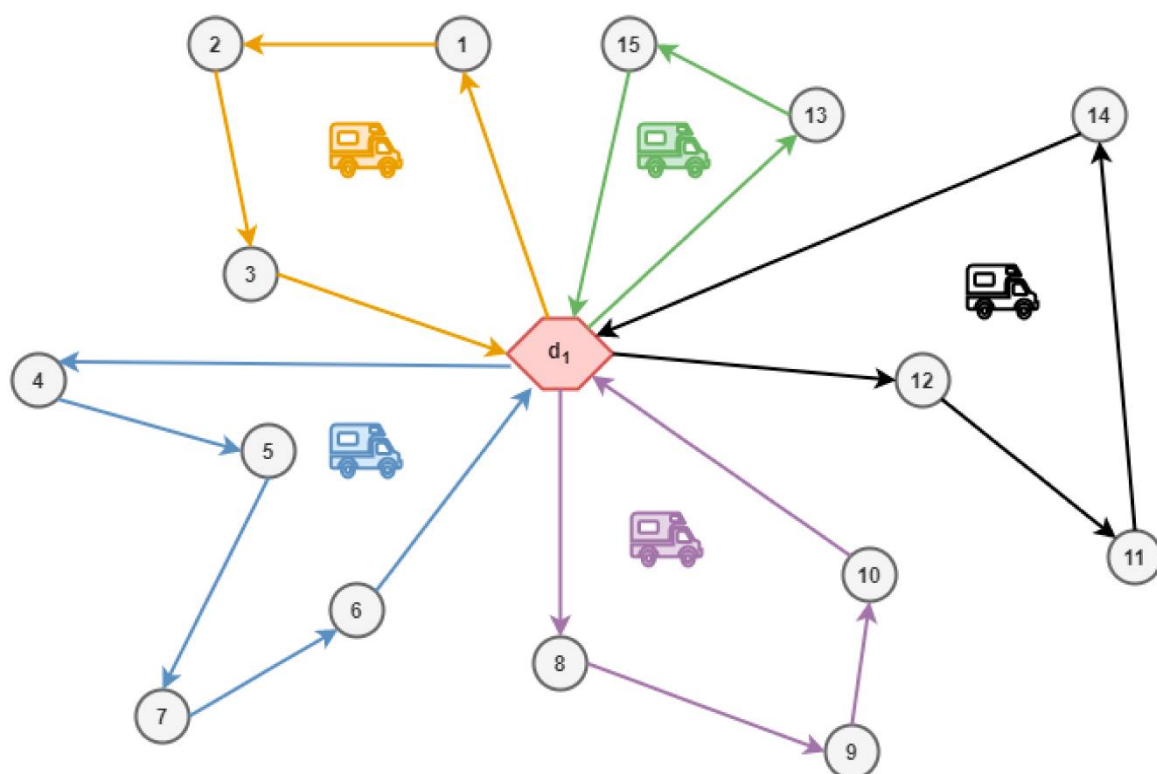
در این سوال از ما خواسته شده است که اقدام به حل کردن مسئله مسیریابی ماشین حمل بار یا همان Vehicle Routing Problem (VPR) با استفاده از الگوریتم کلونی مورچگان بکنیم. هدف این مسئله این است که اقدام به پیدا کردن کوتاهترین مسیر و مسافت برای سرویس دادن به n مشتری با استفاده از m ماشین حمل بار بکنیم.

اگر بخواهیم به زبان ریاضی گسسته به این مسئله نگاه کنیم در حقیقت این مسئله متشکل از یک گراف جهت دار کامل با n راس است که یال‌های این گراف نشان‌دهنده یک مسیر مستقیم بین دو راس است و وزن این یال‌ها حاوی مسافت آن مسیر است. همچنین یکی از راس‌های این گراف نشان‌دهنده انبار مرکزی است و مابقی راس‌ها نیز نشان‌دهنده مشتری هستند.

همچنین به این نکته نیز خوب است اشاره کنم ظرفیت ماشین‌های حمل بار نیز برابر با Q است. نحوه محاسبه وزن هر یال نیز به وسیله فاصله اقلیدسی بین مختصات دو راس مورد محاسبه قرار می‌گیرد. اکنون ما باید کوتاهترین مسافت کلی طی شده توسط m ماشین را با شروط زیر محاسبه کنیم:

- هر مشتری باید تنها یک بار وسط ماشین حمل بار مشخص سرویس‌دهی شود.
- هر ماشین حمل بار باید مسیر خود را از انبار شروع کند و به آن نیز ختم کند.
- مجموع تقاضاهایی که یک ماشین حمل بار پاسخ می‌دهد نباید از Q (ظرفیت ماشین‌های حمل بار) بیشتر شود.

به یک نکته نیز باید توجه کرد که در این حالت بیان شده از مسئله Vehicle Routing Problem ممکن است که همه‌ی مشتریان به صورت کامل سرویس‌دهی نشوند.



شکل 1: نمونه‌ای از نحوه سرویس دهی در مسئله VRP

برای حل این مسئله با روش کلونی مورچه نیاز است که هر مورچه‌ای که ما در نظر می‌گیریم وظیفه یک ماشین حمل بار را شبیه‌سازی کند. و این کار را بدین صورت انجام می‌دهیم که هر مورچه مسیر خود را از انبار شروع می‌کند و راس بعدی را از لیست راس‌های ممکن انتخاب می‌کند و بدین صورت ظرفیت خود را بر روزرسانی می‌کند و این کار را تا جایی انجام می‌دهد که یا همه‌ی راس‌ها را ملاقات کند و یا این که تقاضاها از ظرفیتش بیشتر شده باشد، که در حال دوم مجبور است که به انبار بازگردد. جواب مسئله هنگامی که m مورچه تعیین شده دور خود را تکمیل کردند، محاسبه می‌شود و سپس فرمون‌ها بر روزرسانی می‌شوند. در نهایت این روند برای چندین بار تکرار می‌شود تا جواب مناسب بدست آید.

حال در این سوال به ما یک فایل data.txt داده شده است که در این فایل ابتدا تعداد ماشین‌های حمل بار و ظرفیت هر یک را مشخص می‌کند و در قسمت بعدی نیز یک لیستی از راس‌های گراف را به ما داده است.

به صورت کلی در روش مبتنی بر بهینه‌سازی کلونی مورچگان ما نمی‌خواهیم که مسئله را به صورت بهینه حل کنیم و به نوعی بهینه Global را بدست بیاوریم بلکه ما می‌خواهیم که یک solution نیمه‌بهینه را

بدست بیاوریم و همچنین برای ما مهم است که این محاسبات در یک زمان معقولی نیز انجام شود و نتیجه در این زمان بدست بیاید.

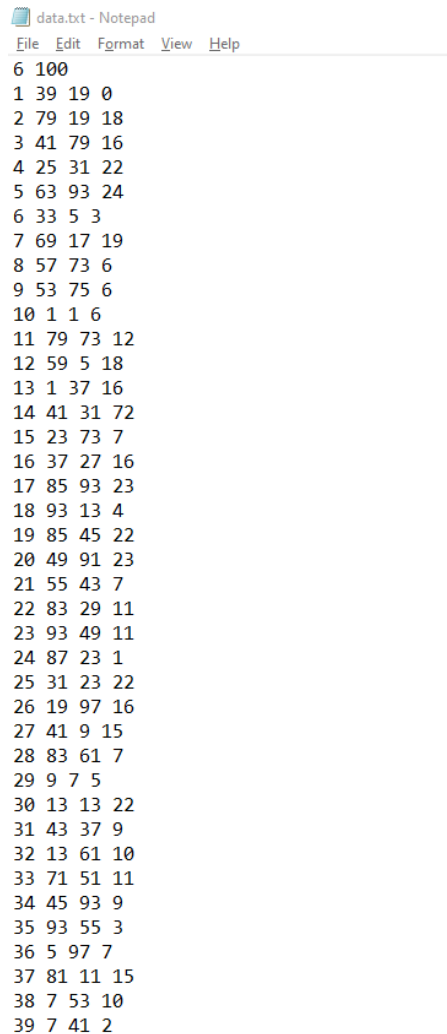
ایده اصلی این روش مورچگان بدین صورت است که مورچه‌هایی که مسیر طولانی‌تری را می‌روند، دیرتر به خانه بازمی‌گردند و بنابراین فرمونی که روی مسیر می‌پاشند، دیرتر روی مورچه تاثیر می‌گذارد و همین اختلاف زمانی که مسیرهای کوتاه‌تر از مسیرهای طولانی‌تر در جذب مورچه‌ها جلو می‌افتند، باعث می‌شود که یک random fluctuation ای ایجاد شود که این random fluctuation با positive feedback پاشیدن فرمون بیشتر تقویت می‌شود و به مرور زمان مسیرهایی که کوتاه‌ترین هستند، بدست مورچه‌ها پیدا می‌شوند.

همچنین خوب است به این نکته نیز قبل از این که وارد پیاده‌سازی شویم اشاره کنم که ما برای مبادله اطلاعات میان مورچه‌ها از Virtual Pheromone استفاده می‌کنیم که بر روی لینک‌ها پاشیده می‌شود. پس لینک (i, j) یک میزان فرمونی دارد که نشان می‌دهد این مسیر چه مقدار مسیر خوبی هست.

این Virtual Pheromone به ما کمک می‌کند که مورچه‌ها اطلاعاتی را که از طریق این دورها کسب می‌کنند را share کنند! و در نهایت به صورت غیرمستقیم Pheromone بر روی مسیر بپاشند. اکنون به سراغ پیاده‌سازی این مسئله می‌رویم:

در چنین مسئله‌ای که ما یک فایل ورودی داریم و می‌خواهیم که اطلاعات مختلفی را از درون آن استخراج کنیم واضح است که مهمترین کار ساخت توابعی بدین منظور است. به همین دلیل در ادامه به انجام این کار می‌پردازیم:

ابتدا من فایل دیتای داده شده را قرار می‌دهم که می‌توانید در ادامه مشاهده کنید:



```
data.txt - Notepad
File Edit Format View Help
6 100
1 39 19 0
2 79 19 18
3 41 79 16
4 25 31 22
5 63 93 24
6 33 5 3
7 69 17 19
8 57 73 6
9 53 75 6
10 1 1 6
11 79 73 12
12 59 5 18
13 1 37 16
14 41 31 72
15 23 73 7
16 37 27 16
17 85 93 23
18 93 13 4
19 85 45 22
20 49 91 23
21 55 43 7
22 83 29 11
23 93 49 11
24 87 23 1
25 31 23 22
26 19 97 16
27 41 9 15
28 83 61 7
29 9 7 5
30 13 13 22
31 43 37 9
32 13 61 10
33 71 51 11
34 45 93 9
35 93 55 3
36 5 97 7
37 81 11 15
38 7 53 10
39 7 41 2
```

شکل 2: فایل دیتای داده شده به ما

توجه: فقط به این نکته توجه شود که ایندکس‌هایی که در شکل بالا مشاهده می‌کنید از شماره 1 شروع شده‌اند تا 39 (یا هر تعداد دیگری که باشد) اما من در پیاده‌سازی‌ام از شماره 0 ایندکس‌ها را داده‌ام و بنابراین در پیاده‌سازی من شماره‌ها یکی کمتر از مقدار موجود در این فایل هست. هرچند که من ایندکس‌ها را نیز ذخیره کرده‌ام و قابل دسترسی هست.

سپس من یک کلاس به نام Vertex می‌سازم که نماینده تک‌تک راس‌های موجود در گراف داده شده به ما هست. در زیر می‌توانید این کلاس را مشاهده کنید:

```

1  class Vertex:
2      def __init__(self, index: int, x_coordinate: float, y_coordinate: float, demand: float):
3          """
4          - Inputs:
5              index: the id or index of the input Vertex
6              x_coordinate: the x axis coordinate of the vertex
7              y_coordinate: the y axis coordinate of the vertex
8              demand: the supplies that customer in this vertex needs!
9          -----
10         - This is the initialization function for Vertex class!
11         -----
12         - Outputs:
13             RETURNS: None
14             - Just update parameters
15         """
16         super()
17         self.index = index
18
19         if demand == 0:
20             self.is_central_warehouse = True
21             self.warehouse_index = index
22         else:
23             self.is_central_warehouse = False
24         self.x_coordinate = x_coordinate
25         self.y_coordinate = y_coordinate
26         self.demand = demand

```

شکل 3: کلاس مربوط به Vertex (هر راس گراف)

سپس در ادامه اقدام به تشکیل یک کلاس دیگر به نام VehicleRoutingProblemGraph می‌کنیم:

```

34 class VehicleRoutingProblemGraph:
35     def __init__(self, input_file_path, Rho=0.1):
36         ...
37         -Inputs:
38             - input_file_path: The file that we want to read info from!
39             - Rho: Rho value! used for virtual pheremone evaporation!
40         -----
41         - This initialization function basically sets the init value for out graph!
42         -----
43         - Outputs:
44             None! : only sets the init values!
45         ...
46         super()
47         self.number_of_nodes = 0
48         self.graph_nodes = None
49         self.number_of_vehicle = 0
50         self.capacity_of_vehicle = 0
51         # this function reads info from file and sets the init values from it!
52         self.initialize_parameters_from_file(input_file_path=input_file_path)
53
54         self.distance_matrix = None
55         # This function sets the distance matrix!
56         self.calculate_distance_matrix()
57         # rho : Pheromone volatilization rate
58         self.Rho = Rho
59
60         # Pheromone Matrix:
61         self.init_pheromone_mat = np.ones((self.number_of_nodes, self.number_of_nodes))
62         self.init_pheromone_mat = 1/(self.init_pheromone_mat * self.number_of_nodes)
63
64         self.virtual_pheromone_matrix = np.ones((self.number_of_nodes,
65             self.number_of_nodes)) * self.init_pheromone_mat
66         # Set the index of the warehouse:
67         self.warehouse_index = 0
68         self.get_warehouse_index()

```

شکل 4: کلاس مربوط به گراف VRP

در شکل بالا همان طور که می‌توانید مشاهده کنید، در تابع Initialization اقدام به مقداردهی اولیه به اکثر پارامترهای موجود در این گراف می‌کنم. همچنین ماتریس فاصله و ماتریس مربوط به فرمون و... را تنظیم می‌کنم و همچنین index مربوط به انبار مرکزی را تنظیم می‌کنم.

در شکل بالا تنها پارامتری که به نظر می‌رسد نیاز به توضیح داشته باشد Rho هست که به صورت زیر برای به روزرسانی و تبخیر شدن virtual pheromone در طول زمان به کار می‌رود:

$$\tau \leftarrow (1-\rho) \tau$$

در ادامه تابع‌های استفاده شده در این کلاس مربوط به گراف را مورد بررسی قرار می‌دهیم:

تابع زیر برای تهیه یک کپی از روی گراف استفاده می‌شود:

```
35 def copy(self, init_pheromone_val=None):
36     new_graph = copy.deepcopy(self)
37     if not None:
38         new_graph.init_pheromone_val = init_pheromone_mat
39         new_graph.virtual_pheromone_matrix = np.ones((new_graph.number_of_nodes,
40                                                         new_graph.number_of_nodes)) * init_pheromone_mat
41     return new_graph
```

شکل 5: تابع کپی کردن گراف

در ادامه نیز تابع مربوط به خواندن اطلاعات از داخل فایل و وارد کردن آن‌ها به داخل کلاس را می‌توانیم که مشاهده کنیم:

```
43 def initialize_parameters_from_file(self, input_file_path="data.txt"):
44     """
45     - Inputs:
46     |   input_file_path: the path of the input file. (Type: String)
47     |   -----
48     - Outputs:
49     |   number_of_vertex: number of the vertices or nodes that the graph has.
50     |
51     """
52     graph_nodes_list = []
53     with open(input_file_path, 'rt') as input_file:
54         line_number = 1
55         for line in input_file:
56             if line_number == 1:
57                 number_of_vehicle, capacity_of_vehicle = line.split()
58                 number_of_vehicle = int(number_of_vehicle)
59                 capacity_of_vehicle = int(capacity_of_vehicle)
60                 print("Vehicle Num: ", number_of_vehicle)
61                 print("Vehicle Capacity: ", capacity_of_vehicle)
62             elif line_number >= 2:
63                 graph_nodes_list.append(line.split())
64                 line_number += 1
65     number_of_nodes = len(graph_nodes_list)
66     for node in graph_nodes_list:
67         print("Item: ", node)
68     graph_nodes = list(Vertex(int(vertex_item[0]), float(vertex_item[1]),
69                             float(vertex_item[2]), float(vertex_item[3])) for vertex_item in graph_nodes_list)
70
71     self.number_of_nodes = number_of_nodes
72     self.graph_nodes = copy.deepcopy(graph_nodes)
73     self.number_of_vehicle = number_of_vehicle
74     self.capacity_of_vehicle = capacity_of_vehicle
```

شکل 6: تابع خواندن اطلاعات از داخل فایل

در بالا همان‌طور که می‌توانید مشاهده کنید، اقدام به خواندن اطلاعات فایل و دسته‌بندی آن‌ها می‌کند و بدین وسیله پارامترهای مختلف گراف و راس‌های آن و... را مشخص می‌کند.

در ادامه نیز ما تابع بدست آوردن و ساخت ماتریس فاصله را داریم برای مسئله:

```

77 def calculate_distance_matrix(self):
78     '''
79     - Inputs:
80     | None: this function basically uses the values
81     | from the class (with using Self)
82     | -----
83     | - This function is creating a matrix which will represent the
84     | "Euclidean Distance" between the nodes of the Graph!
85     | -----
86     - Outputs:
87     | None: it will simply update the values of the class!
88     | ...
89     distance_matrix = np.zeros((self.number_of_nodes, self.number_of_nodes))
90     for i in range(self.number_of_nodes):
91         node_a = self.graph_nodes[i]
92         distance_matrix[i][i] = 1e-9
93         for j in range(i+1, self.number_of_nodes):
94             node_b = self.graph_nodes[j]
95             distance_matrix[i][j] = VehicleRoutingProblemGraph.calculate_euclidean_distance(node_a, node_b)
96             distance_matrix[j][i] = distance_matrix[i][j]
97     # print("Hello: ", distance_matrix)
98     self.distance_matrix = copy.deepcopy(distance_matrix)

```

شکل 7: تابع ساخت ماتریس فاصله

تابع بالا براساس مختصات ها و ... خوانده شده از داخل فایل اقدام به محاسبه فاصله میان هر دو نود و راس موجود در گراف می کند و بدین شکل اقدام به تشکیل یک ماتریس براساس این فاصله ها می کند. و همچنین خوب است به این نکته نیز اشاره کنم که معیار انتخاب فاصله میان هر دو نود براساس فاصله اقلیدسی است و آن را با استفاده از تابع زیر محاسبه می کنیم:

```

101 @staticmethod
102 def calculate_euclidean_distance(point_A, point_B):
103     ...
104     - Inputs:
105         - point_A: the first point coordinates! (Type: Vertex class)
106         - point_B: the second point coordinates! (Type: Vertex class)
107     -----
108     - This function is going to calculate the "Euclidean Distance" between
109     two given points as parameters of this function!
110     -----
111     - Outputs:
112         - euclidean_distance: the calculated "Euclidean Distance" between
113         the two input point! (Type: Float)
114     ...
115     euclidean_distance = np.linalg.norm((point_A.x_coordinate - point_B.x_coordinate,
116                                         point_A.y_coordinate - point_B.y_coordinate))
117     return euclidean_distance

```

شکل 8: تابع محاسبه فاصله اقلیدسی

همان‌طور که در شکل بالا می‌توانید مشاهده کنید با استفاده از دستور نوشته شده می‌توانیم اقدام به محاسبه فاصله اقلیدسی میان هر دو نود دلخواه در گراف کنیم. و فرمول استفاده شده برای محاسبه فاصله اقلیدسی، نیز به صورت زیر می‌باشد:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

\mathbf{p}, \mathbf{q} = two points in Euclidean n-space

q_i, p_i = Euclidean vectors, starting from the origin of the space (initial point)

n = n-space

شکل 9: فرمول محاسبه فاصله اقلیدسی میان هر دو نقطه

اکنون در ادامه نیز ما تابع‌های مربوط به به‌روزرسانی فرمون را داریم:

تابع اول مربوط به تبخیر شدن فرمون پس از هر Iteration است بدین صورت که پس از طی شدن هر Iteration این تابع اجرا می‌شود و اقدام به کم کردن کسری از مقدار فرمون موجود بر روی لینک‌ها می‌کند:

```
158 ✓ def pheremone_evaporation(self):
159     """
160     - Inputs:
161     |     None
162     -----
163     - This function is basically epeporating some portion of the pheremone!
164     -----
165     - Outputs:
166     |     Nothing! : this function only updates the pheremone Matrix!
167     """
168     self.virtual_pheromone_matrix = (1-self.Rho) * self.virtual_pheromone_matrix
```

شکل 10: تابع مربوط به تبخیر کردن فرمون

در ادامه نیز ما تابع مربوط به به‌روزرسانی و اضافه کردن فرمون را داریم:

این تابع به این صورت کار می‌کند که در هر Iteration پس از پیمودن آن به وسیله این تابع اقدام به اضافه کردن فرمون می‌کند و این کار را به صورت زیر انجام می‌دهد:

```

171 ✓ def global_update_pheromone(self, traveled_path, path_distance):
172     """
173     - Inputs:
174     -----
175     - This function is basically updating the Pheremone Matrix!
176     -----
177     - Outputs:
178     |   Nothing! : this function only updates the pheremone Matrix!
179     """
180     current_index = traveled_path[0]
181 ✓ for next_index in traveled_path[1:]:
182     self.virtual_pheromone_matrix[current_index][next_index] += 1 / path_distance
183     current_index = next_index

```

شکل 11: تابع به روزرسانی فرمون

همچنین در ادامه نیز من تابع مربوط به تنظیم مقدار index مربوط به warehouse را قرار می‌دهم:

```

● 188 ✓ def get_warehouse_index(self):
189     """
190 ✓     - Inputs:
191     |   None
192     -----
193     - This function is basically updating the Pheremone Matrix!
194     -----
195 ✓     - Outputs:
196     |   Nothing! : this function only finds and sets the index of the warehouse!
197     """
198     idx = None
199 ✓ for node in self.graph_nodes:
200 ✓     if node.is_central_warehouse is True:
201     |     idx = node.index
202 ✓ if idx is None:
203     |     print("No index found!")
204 ✓ else:
205     |     print("Index Found: ", idx)
206     |     self.warehouse_index = idx - 1

```

شکل 12: تابع مقداردهی index مربوط به warehouse

سپس اکنون در ادامه اقدام به پیاده‌سازی کلاس مورچه می‌پردازیم!

این کلاس به نام Ant در داخل فایل دیگری به نام ant.py قرار دارد که شامل اطلاعات مربوط به یک مورچه است. حال در ادامه به بررسی این کلاس می‌پردازیم:

```

6  class Ant:
7      def __init__(self, input_graph: VehicleRoutingProblemGraph, start_index=0):
8          super()
9          self.start_index = start_index
10         self.graph = input_graph
11         self.current_index = start_index
12         self.vehicle_load = 0
13         self.travel_path = [start_index]
14
15         self.index_to_visit = None
16
17         self.total_travel_distance = 0

```

شکل 13: ساخت کلاس مربوط به مورچه

می‌توانیم ببینیم که در این کلاس اطلاعاتی مانند ظرفیتی که تاکنون بارزده شده و همچنین ایندکس نودی که از آن شروع کرده است قرار دارد و همچنین مسیری که تاکنون پیموده است نیز در آن قرار دارد.

اکنون در ادامه یکی از توابع استفاده شده در آن را قرار می‌دهم به نام `start_over()`:

```

21  def start_over(self):
22      """
23      - Inputs:
24          None
25      -----
26      - This function is basically resetting all the values of the Ant and
27      get it ready for the next iteration!
28      -----
29      -Outputs:
30          None: it simply updates the values of ant to Zero point!
31      """
32      self.start_index = 0
33      self.current_index = self.start_index
34      self.vehicle_load = 0
35      self.travel_path = [self.start_index]
36
37      self.index_to_visit = None
38      # self.index_to_visit.remove(start_index)
39
40      self.total_travel_distance = 0

```

شکل 14: تابع `start_over` از کلاس `Ant`

همان‌طور که در شکل بالا می‌توانید مشاهده کنید و از اسم تابع نیز مشخص است، این تابع این وظیفه را دارد که مسیر پیموده شده و ظرفیت پر شده مورچه را به نوعی صفر یا reset کند. این تابع در پایان هر iteration از اجرای الگوریتم فراخوانده می‌شود تا این که مقادیر مربوط به iteration قبل را پاک کند و مورچه را برای رفتن به iteration بعدی آماده کند.

تابع بعدی به نام `move_vehicle_to_next_node` وظیفه انتقال ماشین/مورچه به نود بعدی گراف که برای آن انتخاب شده است را به عهده دارد. حال در ادامه می‌توانید که آن را مشاهده کنید:

```
42 ✓ def move_vehicle_to_next_node(self, next_node_index):
43     """
44     - Inputs:
45         next_node_index: the index of the next node that we
46         want to visit! (Type: Int)
47
48     -----
49     - This function is moving the ant/vehicle to the next node that
50     it should visit!
51     -----
52     - Outputs:
53         None: it will return nothing! simply updating some parameters!
54     """
55     # Add the new node to the traveled path.
56     self.travel_path.append(next_node_index)
57     # calculate the current travel distance! & add it to the total!
58     current_distance = self.graph.distance_matrix[self.current_index][next_node_index]
59     self.total_travel_distance += current_distance
60
61     if self.graph.graph_nodes[next_node_index].is_central_warehouse:
62         pass
63     # self.vehicle_load = 0
64     else:
65         # add the demand to the vehicle_load
66         self.vehicle_load += self.graph.graph_nodes[next_node_index].demand
67
68     self.current_index = next_node_index
```

شکل 15: تابع مربوط به انتقال اتومبیل به نود بعدی

تابع بالا تنها کاری که انجام می‌دهد این است که نود جدید را به لیست نودهای پیموده شده اضافه می‌کند و همچنین اقدام به محاسبه و به روزرسانی مسیر پیموده شده تا کنون می‌کند.

در ادامه نیز یک تابع دیگر داریم که مشخص می‌کند آیا اتومبیل ما ظرفیت پذیرش بار بیشتری را دارد یا خیر. در ادامه می‌توانید که این تابع را مشاهده کنید:

```

70 def can_vehicle_load_more(self, next_index) -> bool:
71     """
72     - Inputs:
73         next_index: the index of the next node that we
74         want to visit! (Type: Int)
75     -----
76     - In this function we are checking that can we load
77     more and visit more customers or Not!
78     -----
79     -Outputs:
80         result: tell's us that can we load more or not! (Type: bool)
81     """
82     result = None
83     if self.vehicle_load + self.graph.graph_nodes[next_index].demand > self.graph.capacity_of_vehicle:
84         result = False
85     else:
86         result = True
87     return result

```

شکل 16: تابع بررسی ظرفیت داشتن خودرو برای پاسخگویی به نیاز مشتری

همان‌طور که در شکل بالا می‌توانید مشاهده کنید در صورتی که اتومبیل بتواند که تقاضای مشتری را پاسخ بدهد مقدار True و در غیر این صورت مقدار False را باز می‌گرداند که نشان می‌دهد توانایی پاسخگویی به نیاز این مشتری را ندارد.

توجه: به این نکته در اینجا توجه شود که در مسئله از ما خواسته شده است که کامیون یا اتومبیل با حداکثر ظرفیت بار زده شود و سپس در هر نود سرویس دهد و تخلیه کند تا به صفر برسد یا مقدار باقی‌مانده ظرفیتش برای پاسخگویی به نود بعدی کافی نباشد. اما در اینجا و در پیاده‌سازی انجام شده توسط من به این صورت در نظر گرفته‌ام که در ابتدا مقدار بار زده شده صفر است و تا زمانی که به حداکثر ظرفیت برسد یا این که مشتری وجود نداشته باشد که بتواند کامل محموله‌اش را به اتومبیل بدهد می‌تواند به بار زدن ادامه دهد! در واقع من برعکس در نظر گرفته‌ام اما نتیجه در هر دو حالت یکسان می‌شود و تفاوتی نمی‌کند تنها در پیاده‌سازی این روش به نظر من سراسرتر آمد.

اکنون در ادامه به بیان قسمت نهایی و اصلی الگوریتم می‌پردازیم که مسئول اجرای قسمت اصلی الگوریتم بهینه‌سازی مورچگان است و در واقع پیاده‌سازی الگوریتم در این قسمت انجام می‌شود.

در ادامه می‌توانید کلاس **VRPAntColonyOptimization** که به همین منظور ساخته شده است را مشاهده کنید:

```

15 class VRPAntColonyOptimization:
16     def __init__(self, input_graph: VehicleRoutingProblemGraph, number_of_ants=10,
17                 max_iter=200, beta=5, q0=0.8, alpha=2, show_map_or_plot_results=True):
18         """
19         - Inputs:
20             input_graph: the builded graph from file (Type: VehicleRoutingProblemGraph)
21             number_of_ants: The number of ants that we want to use!
22             max_iter = Maximum iterations that we will use!
23             beta: Beta Parameter!
24             q0: (exploitation / exploration) parameter
25             alpha: alpha Parameter
26             show_map: Wheter or not draw Results!
27
28         -----
29         - This funtion basically initializes parameters of the class!
30         -----
31         - Outputs:
32             Nothing
33         """
34         super()
35         # Set the Input Graph:
36         self.graph = input_graph
37         # Set the Number of Ants:
38         self.number_of_ants = number_of_ants
39         self.optimize number of ants()
40         # max_iter : Maximum Iteration
41         self.max_iter = max_iter
42         # vehicle_capacity :
43         self.max_load = input_graph.capacity_of_vehicle
44         # beta Parameter:
45         self.beta = beta
46         # alpha Parameter:
47         self.alpha = alpha
48         # q0 Parameter: (exploitation / exploration)
49         self.q0 = q0
50         # best path
51         self.best_path_distance = math.inf
52         # Saves the best achieved path so far!
53         self.best_path = None
54         self.best_vehicle_num = None
55         self.best_result_each_iteration = None
56         # Initializing Ants according to the given Number!
57         self.ants = list(Ant(self.graph) for _ in range(self.number_of_ants))
58
59         self.show_map = show_map_or_plot_results

```

شکل 17: تابع initialization مربوط به VRPAntColonyOptimization

تابع بالا مقادیر اولیه مربوط به این تابع را تنظیم می‌کند و پارامتر `show_map_or_plot_results` مشخص کننده این است که آیا ما می‌خواهیم که نقشه مربوط به گراف را در انتها به ما نشان دهد یا این که نمودار مربوط به بهترین پاسخ بدست آمده در هر iteration را برای ما رسم کند و همچنین نکته جالب آن قسمت مشخص شده است که در اینجا ما یک تابع دیگر را فراخوانی می‌کنیم تا اقدام به تنظیم تعداد مورچه‌ها کند!

در زیر می‌توانید این تابع را مشاهده کنید:

```
56 ✓ def optimize_number_of_ants(self):
57     """
58     - Inputs:
59     |     Nothing
60     -----
61     - This function is basically optimizing the number_of_ants
62     based on the number of vehicles.
63     -----
64     -Outputs:
65     """
66
67 ✓     if (self.number_of_ants % self.graph.number_of_vehicle) != 0:
68         self.number_of_ants -= (self.number_of_ants % self.graph.number_of_vehicle)
```

شکل 18: تابع تنظیم تعداد مورچه‌ها

این تابع تنظیم تعداد مورچه به این منظور استفاده می‌شود که در صورتی که تعداد مورچه‌های تعیین شده بخش‌پذیر بر تعداد ماشین‌های مورد نیاز ما نباشد تعداد اضافه را کنار می‌گذاریم! به این دلیل این کار را انجام می‌دهیم زیرا روش انجام الگوریتم به این صورت است که ما در هر مرحله می‌آییم و جمعیت مورچه‌ها را تقسیم به دسته‌ها و بچه‌هایی به اندازه تعداد اتومبیل‌ها می‌کنیم و هر کدام از آن‌ها اقدام به جستجو می‌کنند.

در ادامه نیز تابع مربوط به نمایش نتایج iterationهای مختلف را قرار می‌دهم که به صورت یک نمودار نتایج را چاپ می‌کند:

```
74 def plot_learning_fitness(self):
75     """
76     -----
77     This function will draw for us the fitness plot.
78     -----
79     """
80     plt.figure(figsize=(16, 10))
81     plt.plot([i for i in range(len(self.best_result_each_iteration))],
82             self.best_result_each_iteration, c='dodgerblue')
83     best_res = np.argmin(self.best_result_each_iteration)
84     worst_res = np.argmax(self.best_result_each_iteration)
85     line_init = plt.axhline(y=self.best_result_each_iteration[worst_res], color='r', linestyle='--')
86     plt.text(-0.5, best_res, best_res)#, **text_style)
87     line_min = plt.axhline(y=self.best_result_each_iteration[best_res], color='g', linestyle='--')
88     plt.text(-0.5, worst_res, worst_res)#, **text_style)
89     plt.legend([line_init, line_min], ['Initial Fitness', 'Optimized Fitness'])
90     plt.ylabel('Fitness (Percentage (%))')
91     plt.xlabel('Iteration')
92     plt.show()
```

شکل 19: تابع رسم نمودار نتایج

در ادامه اقدام به شروع برای اجرای الگوریتم می‌پردازیم:

```

94  ✓ def ant_colony_optimization_start_algorithm(self):
95      """
96      - Inputs:
97      |   Nothing
98      -----
99      -
100     -----
101  ✓  -Outputs:
102      |   Nothing!
103      """
104      # Start Ant Colony Opt Algorithm:
105      path_queue_for_figure = Queue()
106      aco_thread = Thread(target=self.ant_colony_optimization, args=(path_queue_for_figure,))
107      aco_thread.start()
108
109      # Show Figure or Not:
110  ✓  if self.show_map:
111      |   figure = VehicleRoutingProblemDrawGraph(self.graph, path_queue_for_figure)
112      |   figure.run()
113      aco_thread.join()
114
115      #
116  ✓  if self.show_map:
117      |   path_queue_for_figure.put(PathMessage(None, None))

```

شکل 20: تابع مربوط به اجرای الگوریتم

تابع بالا همچنین اقدام به تشکیل Thread می‌کند برای این که برنامه را به صورت چند نخه همراه با نمایش شکل در صورتی که البته مشخص کرده باشیم برای ما اجرا کند و به این صورت زمان اجرای برنامه را برای ما کاهش می‌دهد.

حال در ادامه تابع مربوط به خود الگوریتم را قرار می‌دهم:

و این تابع را به دلیل طولانی بودن در چند قسمت من قرار می‌دهم:

```

120 def ant_colony_optimization(self, path_queue_for_figure: Queue):
121     """
122     - Inputs:
123
124     -----
125     - This function runs The Ant Colony Optimization Algorithm!
126     -----
127     -Outputs:
128
129     """
130     start_time_total = time.time()
131     # print("Number of Ants: ", self.number_of_ants)
132     batch_number = int(self.number_of_ants / self.graph.number_of_vehicle)
133     # Set the Maximum Iteration Counter
134     start_iteration = 0
135     each_iteration_best_distasnce = []
136     each_iteration_best_path = []
137     for my_iter in range(self.max_iter):
138         ant_counter = 0
139         iteration_paths_distances = []
140         for batch in range(batch_number):
141             # Iterate through the Ants for Solution!
142             remaining_nodes_to_visit = list(range(0, self.graph.number_of_nodes))
143             remaining_nodes_to_visit.remove(self.graph.warehouse_index)
144             batch_paths_distances = []
145             for k in range(self.graph.number_of_vehicle):
146                 # print("K: ", k)
147                 # Go through All the Customers with Ants:
148                 while len(remaining_nodes_to_visit) is not 0:
149                     # Select the Next Node (Customer) to Visit!
150                     next_index = self.select_next_index(self.ants[ant_counter], remaining_nodes_to_visit)
151
152                     if not self.ants[ant_counter].can_vehicle_load_more(next_index):
153                         next_index = self.select_next_index(self.ants[ant_counter], remaining_nodes_to_visit)
154                     if not self.ants[ant_counter].can_vehicle_load_more(next_index):
155                         next_index = self.select_next_index(self.ants[ant_counter], remaining_nodes_to_visit)
156                     if not self.ants[ant_counter].can_vehicle_load_more(next_index):
157                         next_index = self.graph.warehouse_index
158                     if next_index is not self.graph.warehouse_index:
159                         remaining_nodes_to_visit.remove(next_index)
160                     # print("Remaining to Visit: ", remaining_nodes_to_visit)
161                     # Move the Ant to the next Node:
162                     self.ants[ant_counter].move_vehicle_to_next_node(next_index)
163                     # self.graph.local_update_pheromone(self.ants[ant_counter].current_index, next_index)
164                     if self.graph.graph_nodes[next_index].is_central_warehouse:
165                         if self.ants[ant_counter].total_travel_distance != 0:
166                             batch_paths_distances.append(self.ants[ant_counter].total_travel_distance)

```

شکل 21: قسمت اول الگوریتم پیاده سازی شده

در شکل بالا می‌توانیم مشاهده کنیم که از 4 حلقه تو در تو تشکیل شده است و حلقه آخر که While هست مربوط به هر مورچه یا ماشین است که تا زمانی که ظرفیتش به آن اجازه می‌دهد اقدام به رفتن به شهرهای مختلف کند و هنگامی که ظرفیتش تکمیل شد در حلقه for قبل آن اقدام به انتخاب یک مورچه دیگر از میان m (در مثال ما 6 تا مورچه) باقی‌مانده در بچ فعلی می‌کنیم و پس از اتمام این‌ها اکنون به سراغ بچ بعد یعنی m تا مورچه یا ماشین بعدی موجود در جمعیت می‌رویم و این کار را ادامه می‌دهیم تا تمامی جمعیت را بتوانیم که پوشش دهیم.

پس از اتمام مراحل بالا اکنون به سراغ بیرونی‌ترین حلقه for می‌رویم که این حلقه وظیفه شمارش iterationهای مختلف را دارد و به این صورت عمل می‌کند که به تعداد iterationهای مشخص شده اقدام به اجرای الگوریتم گفته شده بر روی تمامی اعضای جمعیت می‌کند.

حال خود هر مورچه را هم توضیح بدهم که به این صورت عمل می کند که در هر مرحله براساس یک تابع به نام `select_next_index` که پایین تر آن را کامل توضیح می دهیم اقدام به انتخاب نود بعدی که باید به آن برود می کند و سپس چک می کند که آیا این نود جدید انتخاب شده خودرو ما قادر هست که به آن برود (یعنی ظرفیت آن مناسب است یا خیر) و در صورتی که نبود ما برای اطمینان 3 بار دیگر روش قبل را اجرا می کنیم و سه نقطه دیگر را نیز بدست می آوریم تا ببینیم که آیا برای نقاط دیگر هم قادر نیست برود یا تنها نقطه مذکور را نمیتوانست. پس از این کار اگر توانست نود را انتخاب کند به آن نود بعدی می رود و تابع های مربوطه برای به روزرسانی پارامترهای خودرو یا مورچه فراخوانی می شود!

علت این که سه بار نیز اقدام به فراخوانی تابع مربوط به انتخاب نود بعدی در صورتی که مشکلی بود می کنیم را نیز پایین تر کامل توضیح می دهیم ولی همین الان نیز اشاره کنم که یک پارامتر `q0` در قسمت اول الگوریتم داشتیم که به نوعی ضریب مربوط به `exploration` و `exploitation` را تنظیم می کند و بنابراین تابع ما در یکی از حالات نیز اقدام به انتخاب رندم نودها می کند و در این حالت ممکن است که ما بتوانیم که به جوابی برسیم.

در ادامه قسمت دوم الگوریتم را قرار می دهیم:

```

168         if len(remaining_nodes_to_visit) is 0:
169             self.ants[ant_counter].move_vehicle_to_next_node(0)
170             ant_counter+=1
171             batch_paths_distances = np.array(batch_paths_distances)
172             batch_distance = np.sum((batch_paths_distances))
173             iteration_paths_distances.append(batch_distance)
174             best_index_in_iteration = np.argmin(iteration_paths_distances)
175             each_iteration_best_distasnce.append(iteration_paths_distances[best_index_in_iteration])
176             each_iteration_best_path.append(self.ants[int(best_index_in_iteration)].travel_path)
177             if iteration_paths_distances[best_index_in_iteration] < self.best_path_distance:
178                 best_idx_start = best_index_in_iteration * self.graph.number_of_vehicle
179                 best_path = []
180                 for i in range(best_idx_start, best_idx_start+ self.graph.number_of_vehicle):
181                     best_path.append(self.ants[i].travel_path)
182                 self.best_path = best_path
183                 self.best_path_distance = iteration_paths_distances[best_index_in_iteration]
184                 self.best_vehicle_num = self.graph.number_of_vehicle
185                 start_iteration = my_iter
186                 print('\n')
187                 print('[iteration %d]: find a improved path, its distance is %f' % (my_iter, self.best_path_distance))
188                 print('it takes %0.3f second multiple_ant_colony_system running' % (time.time() - start_time_total))
189                 # self.graph.global_update_pheromone(self.best_path, self.best_path_distance)
190                 for ant in self.ants:
191                     self.graph.global_update_pheromone(ant.travel_path, ant.total_travel_distance)
192                     ant.start_over()
193                 self.graph.pheremone_evaporation()
194                 print("=====")
195                 print("| iteration:==> ", my_iter))
196             print('\n')
197             print("=====")
198             print("| Final Best Path distance :==> {0} |".format(self.best_path_distance))
199             print("| Number of Vehicles:==> {0} |".format(self.best_vehicle_num))
200             print("=====")
201             print("====>>> 'Best Path Solution' <<<=====")
202             for ant_count, ant_path in enumerate(self.best_path):
203                 print("| Vehicle '{0}' path:=> {1} |".format(ant_count+1, ant_path))
204             print("=====")
205
206             print('| It takes %0.3f second running |' % (time.time() - start_time_total))
207             print("=====")
208             self.best_result_each_iteration = each_iteration_best_distasnce
209             # Show Path in the Map:
210             if self.show_map:
211                 path_queue_for_figure.put(PathMessage(self.best_path, self.best_path_distance))
212
213             if not self.show_map:
214                 self.plot_leaning_fitness()
215         ...

```

شکل 22: قسمت دوم الگوریتم

در شکل بالا می‌توانید مشاهده کنید که بیشتر کارهایی که انجام می‌شود مربوط به چاپ نتایج و مواردی از این دست است و من خیلی توضیح نمی‌دهم و فقط به این موضوع اشاره کنم که در خط 190 تا 192 پس از طی شدن کامل یک iteration و انجام امور مربوط به تمامی اعضای جمعیت اقدام به پاکسازی مسیرهای مربوط به هر مورچه می‌کنیم تا آن را برای iteration بعدی آماده کنیم. همچنین باید به این موضوع نیز اشاره کنم که در خط 191 اقدام به update کردن فرمون می‌کنیم و این کار را باتوجه به مسیری که هر مورچه پیموده است انجام می‌دهیم.

همچنین در خط 193 نیز اقدام به تبخیر کردن فرمون می‌کنم و این خط به ازای هر iteration اجرا می‌شود و براساس یک ضریبی مقداری از فرمون‌ها تبخیر می‌شود.

همچنین در خط‌های 180 تا 188 نیز ما چک می‌کنیم که نتیجه‌ای که در هر مرحله به دست می‌آید آیا نسبت به نتیجه قبلی بهبودی داشته است یا خیر و در صورتی که بهبود پیدا کرده بود اقدام به ذخیره آن می‌کنیم تا در نهایت به عنوان بهترین نتیجه بدست آمده بتوانیم که گزارش دهیم.

خط‌های پایانی نیز همان‌طور که پیش‌تر گفتیم مربوط به نمایش نمودارها و نتایج بدست آمده است.

اکنون در زیر تابع مربوط به انتخاب نود بعدی را قرار می‌دهیم:

```

216 def select_next_index(self, ant: Ant, remaining_nodes=[]):
217     """
218     - Inputs:
219         ant: The ant that want's to select next Node.
220         remaining_nodes: nodes that are remaining to select!
221     -----
222     - This function will select the best next node for us
223     -----
224     - Outputs:
225         next_node = we will return the next node that we will visit!
226     """
227     current_index = ant.current_index
228     transition_prob = np.power(self.graph.virtual_pheromone_matrix[current_index][remaining_nodes], self.alpha) * \
229         np.power(self.graph.distance_matrix[current_index][remaining_nodes], -1 * self.beta)
230
231     transition_prob = transition_prob / np.sum(transition_prob)
232
233     if np.random.rand() < self.q0:
234         max_prob_index = np.argmax(transition_prob)
235         next_index = remaining_nodes[max_prob_index]
236     else:
237         # Now we will randomly select one of the remaining nodes!
238         next_index = VRPAntColonyOptimization.stochastic_accept(remaining_nodes, transition_prob)
239     return next_index

```

شکل 23: تابع انتخاب نود بعدی

همان‌طور که در شکل زیر می‌توانید مشاهده کنید قسمت مشخص شده با رنگ نارنجی برای ما احتمال جابه‌جایی به نودهای بعدی را براساس ماتریس فاصله و فرمون پاشیده شده محاسبه می‌کند و این کار را طبق فرمول زیر انجام می‌دهد:

$$(\tau_{ij})^{\alpha} (d_{ij})^{-\beta}$$

طبق این فرمول یک ماتریس احتمالاتی‌ای محاسبه می‌شود و حال در ادامه براساس مقدار q_0 ای که مشخص کرده‌ایم مشخص می‌شود که باید کدام یک از قسمت‌های قرمز یا آبی اجرا شوند! در صورتی که قسمت قرمز اجرا شود براساس فرمول گفته شده و احتمال و فرمون‌های پاشیده شده اقدام به انتخاب نود بعدی می‌کنیم و در صورتی که قسمت آبی رنگ انتخاب شود اقدام به انتخاب تقریباً رندم (با توجه به ماتریس احتمالاتی محاسبه شده البته) نود بعدی می‌کنیم.

حال در ادامه اقدام به قرار دادن تابع مربوط به انتخاب رندم یا stochastic می‌کنیم:

```

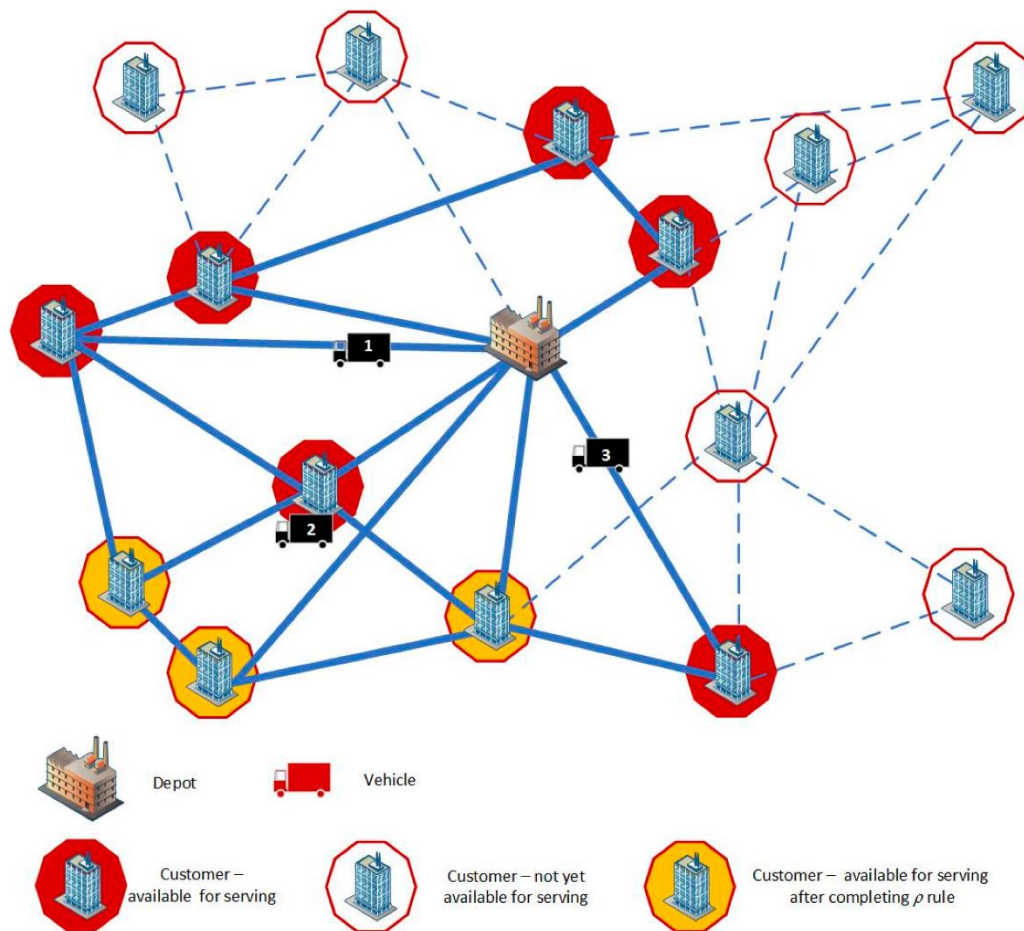
241 @staticmethod
242 def stochastic_accept(index_to_visit, transition_prob): # Randomly select the next Vertex
243     """
244     - Inputs:
245         index_to_visit:
246         transition_prob:
247
248     -----
249     - in this Function we will choose next node randomly!
250     -----
251     -Outputs:
252     """
253     # calculate N and max fitness value
254     N = len(index_to_visit)
255
256     # normalize
257     sum_tran_prob = np.sum(transition_prob)
258     norm_transition_prob = transition_prob / sum_tran_prob
259
260     # select: O(1)
261     while True:
262         # randomly select an individual with uniform probability
263         index = int(N * random.random())
264         if random.random() <= norm_transition_prob[index]:
265             return index_to_visit[index]

```

شکل 24: تابع انتخاب Stochastic نود بعدی

در تابع بالا ما می‌توانیم مشاهده کنیم که براساس تابع احتمالی که محاسبه کرده بودیم اقدام به تولید یک عدد به صورت تصادفی می‌کنیم و براساس تعداد نودهای موجود اقدام به انتخاب نود بعدی‌ای که باید مشاهده کنیم می‌پردازیم.

در شکل زیر نیز به خوبی نحوه کار این الگوریتم را می‌توانید که مشاهده کنید:



شکل 25: نحوه کار الگوریتم

در شکل بالا دقیقاً مواردی را که سعی کردم بالاتر توضیح بدهم را خلاصه کرده است و می‌توانیم ببینیم که در هر مرحله از کارخانه اصلی ماشین‌ها مسیری که می‌پیمایند باعث می‌شود که آن مسیر قدرتمندتر شود به مرور زمان و به تدریج مسیرهای نامناسب‌تر از چرخه انتخاب حذف شوند. و باعث می‌شود که در نهایت مسیرهای نیمه optimum برای ما بدست آید و بتوانیم که از آن‌ها استفاده کنیم.

در ادامه نیز من یک کلاس دیگر که مربوط به رسم نقشه گراف است را قرار می‌دهم و البته خیلی در مورد آن توضیح نمی‌دهم زیرا خیلی مربوط به الگوریتم مسئله و... نیست.


```

8  class VehicleRoutingProblemDrawGraph:
9  def __init__(self, my_graph: VehicleRoutingProblemGraph, path_queue: MPQueue):
10     """
11     matplotlib drawing calculation is placed on the main thread
12
13     """
14     self.my_graph = my_graph
15     self.nodes = my_graph.graph_nodes
16     self.warehouse_index = my_graph.warehouse_index
17     self.figure = plt.figure(figsize=(10, 10))
18     self.figure_ax = self.figure.add_subplot(1, 1, 1)
19     self.path_queue = path_queue
20     self.warehouse_color = 'darkblue'
21     self._customer_color = 'crimson'
22     self._line_color = 'darksalmon'
23     self.line_color_list = ['lime', 'gold',
24                             'deepskyblue', 'orangered', 'magenta', 'blueviolet',
25                             'royalblue', 'lawngreen', 'indigo', 'deeppink',
26                             'darkturquoise', 'springgreen', 'aquamarine', 'darkorange',
27                             'mediumslateblue', 'aqua']

```

شکل 26: تابع Init رسم گراف

در ادامه نیز می‌توانید که تابع مربوط به رسم کردن نقطه‌ها و راس‌های گراف را مشاهده کنید:

```

29  def draw_graph_points(self):
30      print("Warehouse Index: ", self.warehouse_index)
31      self.figure_ax.scatter([self.nodes[self.warehouse_index].x_coordinate],
32                             [self.nodes[self.warehouse_index].y_coordinate],
33                             c=self.warehouse_color, label='Central Warehouse', s=50)
34
35      self.figure_ax.scatter(list(node.x_coordinate for node in self.nodes[1:]),
36                             list(node.y_coordinate for node in self.nodes[1:]),
37                             c=self._customer_color, label='Customer', s=20)
38      plt.pause(0.5)

```

شکل 27: تابع رسم راس‌های گراف

در ادامه نیز تابع run که مربوط به شروع کردن به فرآیند رسم مسیرها هست را می‌توانید که مشاهده کنید:

```

41 def run(self):
42     # Draw Graph Nodes:
43     self.draw_graph_points()
44     # Show the figure:
45     self.figure.show()
46
47     while True:
48         if not self.path_queue.empty():
49             info = self.path_queue.get()
50             while not self.path_queue.empty():
51                 info = self.path_queue.get()
52
53             path, distance, used_vehicle_num = info.get_path_info()
54             if path is None:
55                 print('[draw figure]: exit')
56                 break
57
58             remove_obj = []
59             for line in self.figure_ax.lines:
60                 if line._label == 'line':
61                     remove_obj.append(line)
62
63             for line in remove_obj:
64                 self.figure_ax.lines.remove(line)
65             remove_obj.clear()
66
67             self.figure_ax.set_title('Travel Distance:=> %0.2f, || Number of Vehicles:=> %d ' % (distance, self.my_graph.number_of_vehicle))
68             self.draw_graph_lines(path)
69             plt.pause(1)

```

شکل 28: تابع Run مربوط به رسم مسیرهای گراف

در ادامه نیز می‌توانید که تابع مربوط به رسم خطوط هر یک از مسیرها را مشاهده کنید:

```

71 def draw_graph_lines(self, paths):
72     """
73     """
74     print("Path: ", paths)
75     for path, line_color in zip(paths, self.line_color_list):
76         for i in range(1, len(path)):
77             x_list = [self.nodes[path[i - 1]].x_coordinate, self.nodes[path[i]].x_coordinate]
78             y_list = [self.nodes[path[i - 1]].y_coordinate, self.nodes[path[i]].y_coordinate]
79             self.figure_ax.plot(x_list, y_list, color=line_color, linewidth=2, label='line')
80             plt.pause(0.2)

```

شکل 29: تابع رسم خطوط

❖ سوالات:

1- نحوه فرموله کردن مساله مطابق با روش کلونی مورچه را به همراه تمامی روابط الگوریتم از جمله

نحوه انتخاب راسها، نحوه محاسبه و بروزرسانی فرمون، و تابع هدف مناسب را گزارش نمایید.

این سوال را تقریبا تمامی قسمت‌هایش را به صورت مستقیم و غیر مستقیم در توضیحات قبلی‌ام اگر مطالعه

کنید بیان کردم اما باز هم مختصر توضیحی در این باره می‌دهم.

پیاده‌سازی الگوریتم را همان‌طور که اشاره کردم در قسمت مربوطه و کد آن را نیز کامل توضیح دادم به

این صورت است که ما 4 تا حلقه تو در تو داریم که بالاترین سطح حلقه iteration ها را می‌شمارد و سطح

بعدی اقدام به تقسیم بندی جمعیت به batchها و قسمت‌هایی برابر با تعداد ماشین‌های مجاز یا همان m که در اینجا و مثال داده شده 6 است می‌کند و سپس هر دسته به ترتیب به داخل حلقه بعدی می‌روند و اقدام به مسیریابی می‌کنند و درونی‌ترین حلقه هم مربوط به این است که یک خودرو تا زمانی که گنجایش دارد اقدام به جستجو برای مسیر کند که البته این قسمت نکاتی را داشت که پیش‌تر اشاره کردم و دیگر مجدد نمی‌گوییم.

همچنین برای انتخاب راس‌های همان‌طور که در قسمت مربوطه و پیش‌تر کامل توضیح دادم از دو روش به صورت ترکیبی و با استفاده از یک پارامتر q_0 استفاده کرده‌ام که یک مقدار random تولید شده در هر مرحله با مقایسه خود با q_0 مشخص می‌کند که کدام یک از حالات باید اجرا شود! و حالاتی که داریم عبارت است از حالت انتخاب نود بعدی براساس فرمون‌ها و ماتریس فاصله و فرمول اشاره شده صرفاً یا این که این ماتریس را تبدیل به یک ماتریس احتمالاتی کنیم با نرمال سازی و سپس به صورت رندم (با توجه به مقادیر آرایه‌های این ماتریس و مقایسه آن با یک عدد رندم تولید شده) اقدام به انتخاب نود بعدی کنیم. استفاده از این روش باعث می‌شود که به نوعی خوبی هر دو دنیا را داشته باشیم و هم exploration و هم exploitation خوبی را داشته باشیم.

و در مورد به روزرسانی فرمون هم کامل در قسمت خودش توضیح داده‌ام ولی در اینجا هم بگوییم که براساس فرمولی که در مسئله tsp نیز به کار آمد اقدام به به‌روزرسانی می‌کنیم و به این صورت که مسیرهایی که هر مورچه طی می‌کند را به اندازه 1 بر روی فاصله مقدار فرمونش را افزایش می‌دهیم و همچنین در هر iteration اقدام به تبخیر کسری از فرمون می‌کنیم.

و در مورد تابع هدف با توجه به تمامی توضیحات داده شده تابع هدف ما این است که بتوانیم کوتاهترین مسیر ممکن (نه لزوماً کوتاه‌ترین مسیر واقعی بلکه نزدیک به آن) را در زمان معقولی براس سرویس دهی به کارخانه‌ها انتخاب کنیم.

2- مقادیر بهینه تمامی پارامترهای بکار گرفته شده در مساله را گزارش نمایید و تفسیر خود را از آنها بیان کنید.

در پاسخ به این قسمت من در قسمت نتایج که بعد از این سوالات هست کامل این موارد را توضیح داده‌ام. لطفاً به آنجا مراجعه کنید.

3- چه راهکارهایی را میتوان برای بهتر کردن جواب مساله بکار برد؟ حداقل دو مورد را ذکر نمایید.

پاسخ این قسمت سوال را تا حدودی با کاری که با پیاده‌سازی تابع انتخاب نود بعدی انجام داده‌ام داده‌ام و این مقدار q_0 که باعث افزایش و تنظیم مقدار exploration و exploitation می‌شود بسیار بر روی عملکرد مسئله تاثیر داشته است و نتایج را بسیار بهتر کرده است. همچنین برای بهبود بهتر می‌توان به نوعی مقدار این q_0 را به پویا درآورد یعنی به این صورت که هر چند Iteration یک بار کم یا زیاد شود بسته به نیاز ما که بیشتر به سمت انتخاب تصادفی و exploration برویم یا این که به سمت exploitation برویم.

همچنین می‌توانیم با تغییر جمعیت و افزایش آن اقدام به بهبود نتایج کنیم اما خب ممکن است به بهای زیاد شدن زمان محاسبات ما تمام شود و به نوعی این کار یک trade off است.

همچنین کار مهم دیگری که می‌توان انجام داد استفاده از روش‌های مختلف دیگر برای به روزرسانی فرمون است. روش‌هایی مانند Elitist ant یا Worst Tour یا Greedy update rule.

مثلا در روش Elitist Ant به آن مورچه‌ای که بهترین طول را از ابتدای اجرای الگوریتم تاکنون داشته است اجازه می‌دهیم که علاوه بر مورچه‌های دیگر که اقدام به update کردن فرمون‌هایشان می‌کنند این مورچه هم مجدد همراه با آن‌ها بیاید و Update کند. و به نوعی با انجام این کار بهترین مورچه نسل قبل تجربیات خودش را دارد به نسل‌های جدید منتقل می‌کند و به آن‌ها اجازه می‌دهد که مسیر بهتر فرمونش افزایش پیدا کند تا بقیه نیز آن را انتخاب کنند.

و حالت Worst tour به این صورت عمل می‌کند که ما آن طولی که بدتر از بقیه هست و طولش از همه بیشتر است را اجازه نمی‌دهیم که مورچه‌ها بر روی آن اثر بگذارند و فرمونش را آپدیت کنند.

این کار یک نوع نخبه گرایی هست و ما اجازه نمی‌دهیم که مسیرهای طولانی در جواب ما تاثیر ایجاد کنند.

و نیز روش Greedy update rule همان‌طور که از اسمش پیداست به صورت کاملاً حریصانه می‌گوید که فقط آن مورچه‌ای که بهترین جواب را دارد می‌تواند که فرمونش را آپدیت کند و بقیه اجازه ندارند که به روزرسانی انجام بدهند.

این روش آخر exploitation اش خیلی زیاد است و ممکن است که ما را دچار local optima کند و exploration کمی دارد که باعث می‌شود نتوانیم از این local optima خارج شویم.

اما این روش در مسائلی که محاسبات خیلی زیادی دارد و زمان زیادی طول می کشد می تواند که کاربرد داشته باشد.

همچنین در همین بحث به روزرسانی فرمون یک کار دیگری هم که می توان انجام داد این است که وقتی یک مورچه یک مسیری را طی کرد به مورچه بعدی اجازه ندهیم که مجدد آن مسیر را طی کند تا بدین وسیله exploration را بالا ببریم که البته با توجه به خواسته های مسئله خودمان من عملا این کار را در پیاده سازی ام انجام داده ام. خوبی انجام این کار که البته ما انجام داده ایم این است که باعث می شود که در صورتی که یک مورچه یک مسیری را رفت و خیلی از بقیه بهتر بود به نوعی باعث شود که بقیه همه به سمت این مسیر بیایند و دیگر دنبال مسیرهای دیگر نروند و این حالت در صورتی که اختلاف بین fitness های خیلی زیاد باشد روی می دهد.

4- برای داده های مذکور، علاوه بر کمترین مسافت کلی و مسیر طی شده توسط هر ماشین حمل بار، نمودار کمترین مسافت (جواب مساله) بر حسب تعداد تکرار الگوریتم را به همراه تفسیر خود گزارش نمایید.

من این کار را در ادامه در قسمت نتایج انجام می دهم و تمامی نتایج بدست آمده را قرار می دهم:

❖ نتایج:

اکنون در این قسمت من نتایج بدست آمده از اجرای این الگوریتم و برنامه را قرار می دهم:

برای اجرای الگوریتم کافی است که فایل زیر را با پایتون اجرا کنید:

ant.py	2021-06-04 10:32 PM	Python Source File	4 KB
Build_up_Graph.py	2021-06-04 10:31 PM	Python Source File	10 KB
Build_up_Graph_Map.py	2021-06-04 10:41 PM	Python Source File	4 KB
data.txt	2019-04-27 1:58 PM	Text Document	1 KB
Main_ACO_Algorithm.py	2021-06-04 10:38 PM	Python Source File	14 KB

شکل 30: فایل اصلی

و این فایل های در پوشه code فایل آپلود شده قرار دارند.

همچنین برای تغییر پارامترهای اولیه مسئله نیز می توانید از طریق همین فایل اقدام کنید که در ادامه من به آن ها اشاره می کنم:

```

269 if __name__ == '__main__':
270     file_path = 'data.txt'
271     ants_num = 100
272     beta = 2
273     alpha = 0.5
274     q0 = 0.1
275     show_figure = True
276     max_iteration = 300
277     graph = VehicleRoutingProblemGraph(file_path)
278     macs = VRPantColonyOptimization(input_graph=graph, max_iter=max_iteration, number_of_ants=ants_num,
279                                   beta=beta, alpha=alpha, q0=q0, show_map_or_plot_results=show_figure)
280     macs.ant_colony_optimization_start_algorithm()

```

شکل 31: تنظیم پارامترها

توجه: برای رسم Map مربوط به صفحه گراف مقدار `show_figure` را بر روی `True` قرار دهید و برای رسم نمودار مربوط به تغییرات پاسخ در `iteration`های مختلف مقدار آن را برابر با `false` قرار دهید.

در شکل زیر نیز می‌توانید که مسیرهای بدست آمده برای این اجرا را مشاهده کنید و همچنین بهترین جواب بدست آمده و نیز زمان اجرای الگوریتم:

```

=====
Final Best Path distance :==> 646.0222114393914 |
Number of Vehicles:==> 6 |
=====
>>> 'Best Path Solution' <<<=====
Vehicle '1' path:=> [0, 3, 38, 12, 28, 9, 29, 37, 31, 14, 0] |
Vehicle '2' path:=> [0, 26, 5, 15, 0] |
Vehicle '3' path:=> [0, 24, 13, 0] |
Vehicle '4' path:=> [0, 11, 6, 1, 23, 21, 18, 22, 0] |
Vehicle '5' path:=> [0, 30, 20, 8, 7, 4, 19, 33, 2, 0] |
Vehicle '6' path:=> [0, 32, 27, 10, 16, 25, 35, 17, 36, 34, 0] |
=====
It takes 19.828 second running |
=====

```

شکل 32: پاسخ‌های بدست آمده برای این اجرا

همچنین قسمتی از مراحل اجرای الگوریتم را نیز در زیر می‌توانید مشاهده کنید:

```

[iteration 0]: find a improved path, its distance is 937.667058
it takes 0.206 second multiple_ant_colony_system running
=====
| iteration==> 0
=====
| iteration==> 1
=====

[iteration 2]: find a improved path, its distance is 922.198962
it takes 0.359 second multiple_ant_colony_system running
=====
| iteration==> 2
=====

Warehouse Index: 0
[iteration 3]: find a improved path, its distance is 910.785849
it takes 0.472 second multiple_ant_colony_system running
=====
| iteration==> 3
=====
| iteration==> 4
=====
| iteration==> 5
=====

[iteration 6]: find a improved path, its distance is 878.208507
it takes 0.775 second multiple_ant_colony_system running
=====
| iteration==> 6
=====

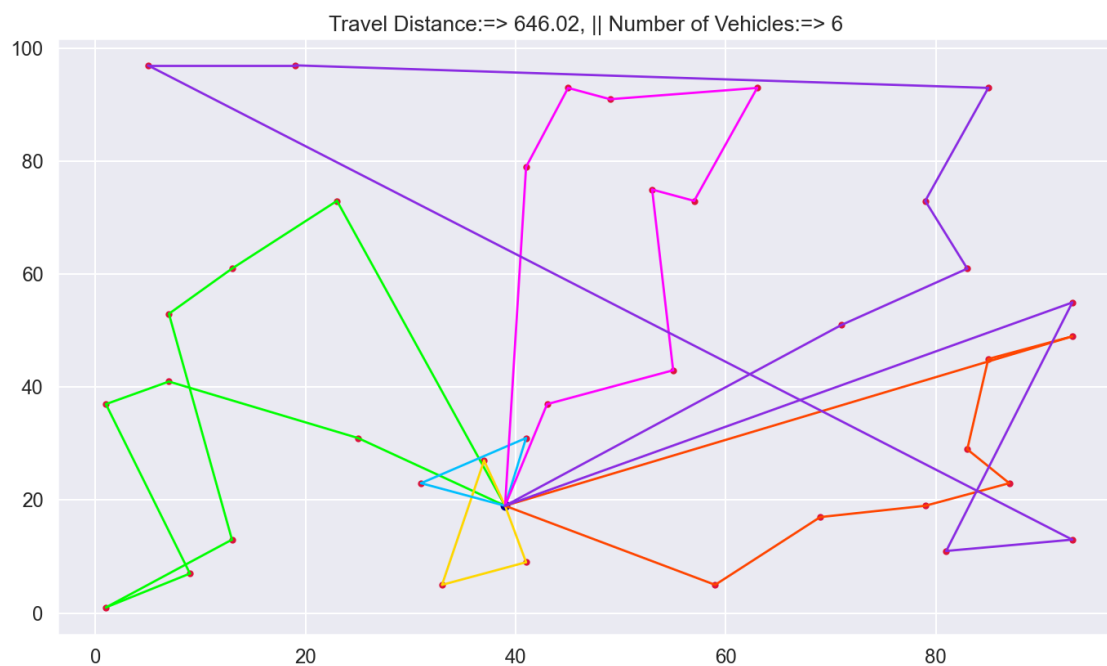
[iteration 7]: find a improved path, its distance is 863.822746
it takes 0.877 second multiple_ant_colony_system running
=====
| iteration==> 7
=====
| iteration==> 8
=====
| iteration==> 9
=====

[iteration 10]: find a improved path, its distance is 856.548221
it takes 1.190 second multiple_ant_colony_system running
=====
| iteration==> 10
=====

```

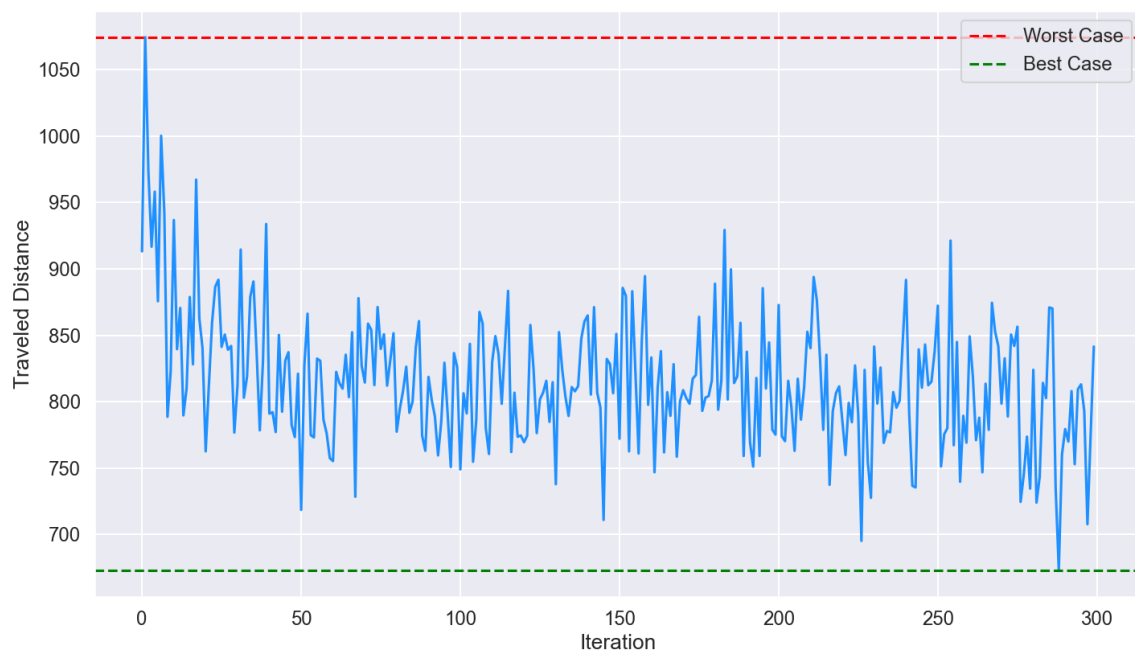
شکل 33: قسمتی از اجرای الگوریتم

در ادامه نیز می‌توانید نقشه مربوط به گراف بدست آمده را مشاهده کنید:



شکل 34: مسیرهای طی شده در بهترین پاسخ برای پاسخ‌گویی به مشتریان

در ادامه نیز می‌توانید که نمودار مربوط به نتایج در Iteration‌های مختلف را پیدا کنید:



شکل 35: نمودار بهترین فاصله‌های بدست آمده از میان جمعیت در هر Iteration در این حالت

همان‌طور که در شکل بالا می‌توانیم مشاهده کنیم مرتب پاسخ‌ها در حال بهبود بوده‌اند تا به نقطه بهترین رسیده است. و بنابراین الگوریتم دارد به خوبی کار می‌کند.

در مورد پارامترهای تنظیم شده در این قسمت بگویم که من مقدارهای تنظیم شده را به ازای بسیاری از مقادیر تست کرده‌ام که دیگر در اینجا نمی‌آورم اما به ازای این مقادیر توانستم که بهترین جواب‌ها را بگیرم. همچنین خوب است به این موضوع هم اشاره کنم که در اجراهای پشت سرهم که بگیریم نتایج کمی تا قسمتی با هم فرق میکند.

تعداد مورچه یا همان جمعیت 100 به ما جستجوی خوبی را در هر Iteration می‌دهد و به علاوه خیلی از ما زمان بیشتری نمی‌گیرد و در صورتی که مقدار را بیشتر از 100 در نظر بگیریم الگوریتم ما خیلی طولانی و زمان‌بر می‌شود که از هدف اصلی الگوریتم کلونی مورچگان که رسیدن به پاسخ نیمه‌بهینه در زمان معقول بود دور می‌شود.

همچنین q_0 را مقدارش را برابر با 0.1 گرفتم که میزان exploration ما بالاتر باشد و در این حالت توانستم جواب‌های بهتری را در مقایسه با حالتی که مقدار q_0 بیشتر بود بدست بیاورم.

پارامتر α را نیز مقدار کوچکی در نظر گرفتم زیرا مقادیر موجود در ماتریس مربوط به فرمون کوچک هست (کمتر از صفر) و با در نظر گرفتن مقدار کوچکتر برای α باعث شده‌ام که اثر فرمون‌ها بیشتر شود.

همچنین پارامتر Beta را نیز یک مقدار معقول یعنی 2 در نظر گرفتم البته مقادیر بیشتر هم جواب خوبی میداد و خیلی نتایج تغییری نمی‌کرد اما با 2 توانستم بهترین جواب را دریافت کنم و بنابراین به همین مقدار اکتفا کردم.

همچنین پارامتر Rho را نیز برابر با 0.1 در نظر گرفتیم که باعث می‌شود این پارامتر که در هر Iteration مقدار فرمون ما در 0.9 ضرب شود و به عبارتی 10 درصد کاهش پیدا کند که مقدار معقولی هست.

در ادامه نیز نتایج مربوط به تعداد Iteration برابر با 1000 را نیز قرار می‌دهم:

ابتدا پارامترها را در این حالت مشاهده کنید:

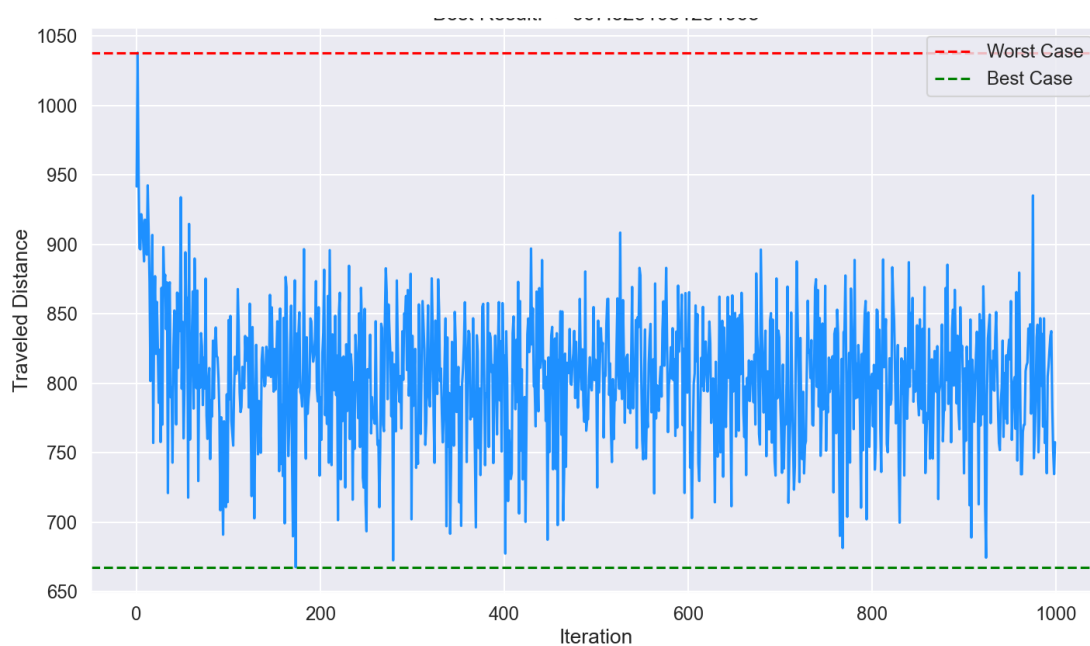
```

270 if __name__ == '__main__':
271     file_path = 'data.txt'
272     ants_num = 100
273     beta = 2
274     alpha = 0.5
275     q0 = 0.1
276     show_figure = False
277     max_iteration = 1000
278     graph = VehicleRoutingProblemGraph(file_path)
279     macs = VRPAntColonyOptimization(input_graph=graph, max_iter=max_iteration, number_of_ants=ants_num,
280                                   beta=beta, alpha=alpha, q0=q0, show_map_or_plot_results=show_figure)
281     macs.ant_colony_optimization_start_algorithm()

```

شکل 36: پارامترهای تنظیم شده در حالت iteration 1000

در ادامه می‌توانید که نمودار مربوط به بهترین نتیجه در هر Iteration را مشاهده کنید:



شکل 37: بهترین نتیجه در هر Iteration

همچنین در ادامه نیز می‌توانید مسیرها و زمان بدست آمده را مشاهده کنید:

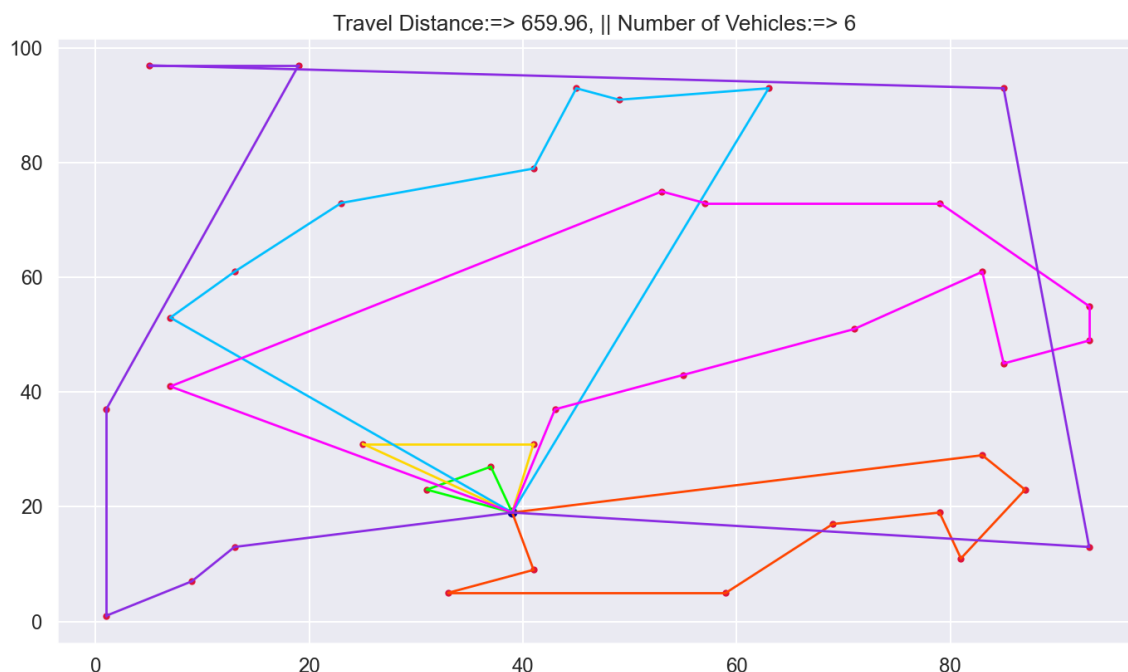
```

=====|
| Final Best Path distance :==> 659.9599015835115 |
| Number of Vehicles:==> 6 |
|=====|
|=====>>> 'Best Path Solution' <<<=====|
| Vehicle '1' path:=> [0, 24, 15, 0] |
| Vehicle '2' path:=> [0, 3, 13, 0] |
| Vehicle '3' path:=> [0, 37, 31, 14, 2, 33, 19, 4, 0] |
| Vehicle '4' path:=> [0, 26, 5, 11, 6, 1, 36, 23, 21, 0] |
| Vehicle '5' path:=> [0, 30, 20, 32, 27, 18, 22, 34, 10, 7, 8, 38, 0] |
| Vehicle '6' path:=> [0, 29, 28, 9, 12, 25, 35, 16, 17, 0] |
|=====|
| It takes 62.314 second running |
|=====|
| Path: [[0, 24, 15, 0], [0, 3, 13, 0], [0, 37, 31, 14, 2, 33, 19, 4, 0], [0, 26, 5, 11, 6, 1, 36, 23, 21, 0], [0, 30, 20, 32, 27, 18, 22, 34, 10, 7, 8, 38, 0], [0, 29, 28, 9, 12, 25, 35, 16, 17, 0]]

```

شکل 38: نتایج بدست آمده در این حالت

در ادامه نیز می‌توانید که نقشه رسم گراف را مشاهده کنید:



شکل 39: گراف و مسیرهای بدست آمده در این حالت

همان‌طور که می‌توانید مشاهده کنید در این حالت تغییر خیلی چشمگیری نسبت به حالت‌های قبل نداشته‌ایم و بنابراین ارزش این که 1000 ایتريشن را طی کنیم واقعا حس نمی‌شود و کار بی‌هوده‌ای به نظر می‌آید.

در ادامه نیز می‌توانید که قسمتی از اجرای این برنامه را نیز مشاهده کنید:

```

[iteration 0]: find a improved path, its distance is 998.276766
it takes 0.203 second multiple_ant_colony_system running
=====
| iteration:==> 0
=====
[iteration 1]: find a improved path, its distance is 933.984853
it takes 0.290 second multiple_ant_colony_system running
=====
| iteration:==> 1
=====
| iteration:==> 2
=====
[iteration 3]: find a improved path, its distance is 885.785792
it takes 0.452 second multiple_ant_colony_system running
Warehouse Index: 0
=====
| iteration:==> 3
=====
[iteration 4]: find a improved path, its distance is 873.812494
it takes 0.525 second multiple_ant_colony_system running
=====
| iteration:==> 4
=====
| iteration:==> 5
=====
| iteration:==> 6
=====

[iteration 7]: find a improved path, its distance is 873.651474
it takes 0.835 second multiple_ant_colony_system running
=====
| iteration:==> 7
=====
| iteration:==> 8
=====

[iteration 9]: find a improved path, its distance is 832.235368
it takes 1.005 second multiple_ant_colony_system running
=====
| iteration:==> 9
=====
| iteration:==> 10
=====
| iteration:==> 11
=====

```

```

259         norm_transition_prob = np.sum(transition_prob, axis=0)
260         norm_transition_prob = norm_transition_prob / norm_transition_prob.sum()
261         # select: O(1)
262         # roulette wheel selection
263         # randomly select
264         index = int(N * random.random())
265         if random.random() < norm_transition_prob[index]:
266             return index
267
268
271         file_path = 'data.txt'
272         ants_num = 100
273         beta = 2
274         alpha = 0.5
275         show_figure = True
276         max_iteration = 300
277         graph = VehicleRoutingProblem()
278         macs = VRPAntColonyOptimization()
279         macs.set_graph(graph)
280         macs.set_ants_num(ants_num)
281         macs.ant_colony_optimization()

```

شکل 40: قسمتی از اجرای برنامه در این حالت

توجه شود که تمامی فایل‌های کدها در پوشه Code در فایل Upload شده قرار دارد.

باتشکر از زحمات شما