C1M3_Assignment

October 8, 2024

1 Assignment - Implementing graph algorithms with LLMs

Welcome to the first assignment of Course 1 in the Generative AI for Software Development Specialization!

1.1 1 - Introduction

In this assignment, you'll start by working with a basic `Graph` class that comes with simple methods. Then, you'll move on to another class named `Graph_Advanced`. Here, you'll be tasked with implementing four methods to tackle four different problems.

It's important to note that the Graph class cannot be modified in any way. However, you're free to add as many methods as you need to the `Graph_Advanced` class to address the challenges you'll encounter.

We expect you to leverage an LLM (Large Language Model) to aid in developing the algorithms. These algorithms should not only aim for solutions that are close to optimal but also adhere to a specific time execution limit. We provide an LLM with GPT-3.5 for you, the link is here, but feel free to use the LLM you want!

LINK FOR GPT-4o

Don't worry if things seem a bit unclear at the moment! You'll receive guidance throughout the assignment.

We expect you to use LLMs in the following manner:

- Begin by clearly defining the problem you are trying to solve or the concept you wish to understand better. Present this to the LLM in a concise manner.
- Engage with the LLM as if it were a knowledgeable peer. Ask questions, seek clarifications, and propose hypotheses for the LLM to comment on.
- Provide the LLM the Graph class given to you and ask it to explain how it works.
- After receiving input from the LLM, critically evaluate the information and decide how best to apply it to your work. Not all suggestions from the LLM may be directly applicable or correct; part of your task is to discern the most valuable advice.
- **NOTE**: DO **NOT** use joblib or any parallel computing technique as it will break the autograder. If necessary explicitily tell the LLM to avoid such algorithms. You also should

NOT install new libraries. If you want to use a library not imported here (with exception of joblib and multiprocessing or any other for parallel computing), please do so in the Graph_Advanced cell.

TIPS FOR SUCCESSFUL GRADING OF YOUR ASSIGNMENT:

- All cells are frozen except for the ones where you need to submit your solutions.
- You can add new cells to experiment but these will be omitted by the grader, so don't rely on newly created cells to host your solution code, use the provided places for this.
- To submit your notebook, save it and then click on the blue submit button at the beginning of the page.

1.1.1 1.1 Importing necessary libraries

Let's begin by importing the necessary libraries. **It's crucial to remember that you should not use any libraries beyond those imported for this assignment. Please communicate this constraint clearly to the LLM. If you need to import a library not imported in the cell below, please import it in the cell containint the Graph_Advanced class, anywhere else, the autograder won't be able to parse it, thus making the grade fail.**

[1]:
```
import random
import heapq
import os
import numpy as np
import itertools
```

1.1.2 1.2 Importing unittests library

This library includes basic unittests to evaluate your solution. If you don't pass some of these unittests, it's probable that your solution will also fail upon submission. However, passing all the unittests doesn't guarantee success in this assignment, as the autograder will subject your solutions to more extensive testing. To successfully pass this assignment, your solution needs to meet the requirements in more than 85% of the test cases. If your solution fails, you will receive up to three examples highlighting where your algorithm fell short, along with instructions on how to replicate the graphs for debugging purposes.

[2]:
```
import unittests
```

Here is the Graph class you'll be working with. Please take a moment to familiarize yourself with it and its methods. Feel free to ask an LLM to help explain how the class functions!

[3]:
```
class Graph:
    def __init__(self, directed=False):
        """
        Initialize the Graph.

        Parameters:
```

```
        - directed (bool): Specifies whether the graph is directed. Default is␣
↪False (undirected).

        Attributes:
        - graph (dict): A dictionary to store vertices and their adjacent␣
↪vertices (with weights).
        - directed (bool): Indicates whether the graph is directed.
        """
        self.graph = {}
        self.directed = directed

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.

        Parameters:
        - vertex: The vertex to add. It must be hashable.

        Ensures that each vertex is represented in the graph dictionary as a␣
↪key with an empty dictionary as its value.
        """
        if not isinstance(vertex, (int, str, tuple)):
            raise ValueError("Vertex must be a hashable type.")
        if vertex not in self.graph:
            self.graph[vertex] = {}

    def add_edge(self, src, dest, weight):
        """
        Add a weighted edge from src to dest. If the graph is undirected, also␣
↪add from dest to src.

        Parameters:
        - src: The source vertex.
        - dest: The destination vertex.
        - weight: The weight of the edge.

        Prevents adding duplicate edges and ensures both vertices exist.
        """
        if src not in self.graph or dest not in self.graph:
            raise KeyError("Both vertices must exist in the graph.")
        if dest not in self.graph[src]:  # Check to prevent duplicate edges
            self.graph[src][dest] = weight
        if not self.directed and src not in self.graph[dest]:
            self.graph[dest][src] = weight

    def remove_edge(self, src, dest):
        """
        Remove an edge from src to dest. If the graph is undirected, also␣
↪remove from dest to src.

        Parameters:
        - src: The source vertex.
        - dest: The destination vertex.
        """
        if src in self.graph and dest in self.graph[src]:
            del self.graph[src][dest]
        if not self.directed:
            if dest in self.graph and src in self.graph[dest]:
                del self.graph[dest][src]

    def remove_vertex(self, vertex):
        """
        Remove a vertex and all edges connected to it.

        Parameters:
        - vertex: The vertex to be removed.
        """
        if vertex in self.graph:
            # Remove any edges from other vertices to this one
            for adj in list(self.graph):
                if vertex in self.graph[adj]:
                    del self.graph[adj][vertex]
            # Remove the vertex entry itself
            del self.graph[vertex]

    def get_adjacent_vertices(self, vertex):
        """
        Get a list of vertices adjacent to the specified vertex.

        Parameters:
        - vertex: The vertex whose neighbors are to be retrieved.

        Returns:
        - List of adjacent vertices. Returns an empty list if vertex is not␣
↪found.
        """
        return list(self.graph.get(vertex, {}).keys())

    def _get_edge_weight(self, src, dest):
        """
        Get the weight of the edge from src to dest.

        Parameters:
        - src: The source vertex.
        - dest: The destination vertex.

        Returns:
        - The weight of the edge. If the edge does not exist, returns infinity.
        """
        return self.graph[src].get(dest, float('inf'))

    def __str__(self):
        """
        Provide a string representation of the graph's adjacency list for easy␣
↪printing and debugging.

        Returns:
        - A string representation of the graph dictionary.
        """
        return str(self.graph)
```

You are required to modify a new class, which inherits from the Graph class, by filling in the blank methods listed below. It's entirely acceptable, and often necessary, to add auxiliary (helper) methods to address each problem effectively.

However, the primary function tasked with solving each problem must retain the names provided below and return the output in the specified order: first the distance, followed by the path. Variable names in the return call are just examples. You can change them, as long as the first argument is the distance and the second is the path, given by a list of node numbers.

1.2 Example of an expected function output

```
graph = Graph_Advanced() # Assume the graph is already built with nodes and edges

print(graph.shortest_path(5, 260)) # Find the shortest path between node 5 and 260.

310, [5,17, 80, 71, 190, 260]
```

1.3 2 - The Graph_Advanced Class

Below is the Graph_Advanced class, which is a subclass of the Graph class. You are tasked with implementing 4 methods, each corresponding to an exercise in this assignment. It's crucial to remember that you **must not** alter the names of these functions, and the output format must be (distance, path) as specified. You **can** introduce auxiliary methods **within the Graph_Advanced class** to aid in solving any of the exercises, but the final solution **must** be delivered through the functions listed below.

1.3.1 2.1 How to Complete The Exercises with LLM Assistance

1. **Problem Understanding with LLM**: Discuss graph theory problems like the Travelling Salesman Problem (TSP) and shortest path finding with the LLM. Use it to clarify concepts and get examples, enhancing your foundational knowledge.

2. **Graph Class Analysis with LLM**: Analyze the provided Graph class functions and structure by consulting the LLM. Its explanations will help you understand how to effectively utilize and extend the class.
3. **Method Implementation Guidance**: For coding tasks (`shortest_path`, `tsp_small_graph`, `tsp_large_graph`, `tsp_medium_graph`), brainstorm with the LLM. Share your logic and seek optimization tips to refine your solutions. You can even ask the LLM to build the code for you, with any specification you want.
4. **Developing Helper Methods**: Before adding helper methods to the `Graph_Advanced` class, discuss your ideas with the LLM for feedback on implementation strategies, ensuring they align with the class's integrity.
5. **Testing Solutions with LLM**: Use the LLM to strategize and review your testing approach, especially for tests that yield unexpected results. It can offer debugging and optimization advice to improve your solutions.

Keep the interaction with the LLM continuous, treating it as a mentor for advice, debugging, and review throughout the exercise process. This approach will not only aid in task completion but also in skill enhancement.

[22]:
```
class Graph_Advanced(Graph):

    def shortest_path(self, start, end):
        """
        Calculate the shortest path from a starting node to an ending node in a␣
↪sparse graph
        with potentially 10000s of nodes. Must run under 0.5 second and find␣
↪the shortest distance between two nodes.

        Parameters:
        start: The starting node.
        end: The ending node.

        Returns:
        A tuple containing the total distance of the shortest path and a list␣
↪of nodes representing that path.
        """
        # Dijkstra's algorithm
        queue = [(0, start, [])]
        seen = set()
        min_dist = {start: 0}

        while queue:
            (cost, v1, path) = heapq.heappop(queue)
            if v1 in seen:
                continue

            seen.add(v1)
            path = path + [v1]

            if v1 == end:
                return (cost, path)

            for v2, weight in self.graph.get(v1, {}).items():
                if v2 in seen:
                    continue
                prev = min_dist.get(v2, None)
                next = cost + weight
                if prev is None or next < prev:
                    min_dist[v2] = next
                    heapq.heappush(queue, (next, v2, path))

        return (float('inf'), [])

    def tsp_small_graph(self, start_vertex):
        """
        Solve the Travelling Salesman Problem for a small (~10 node) complete␣
↪graph starting from a specified node.
        Required to find the optimal tour. Expect graphs with at most 10 nodes.␣
↪Must run under 0.5 seconds.

        Parameters:
        start: The starting node.

        Returns:
        A tuple containing the total distance of the tour and a list of nodes␣
↪representing the tour path.
        """
        vertices = list(self.graph.keys())
        vertices.remove(start_vertex)
        n = len(vertices)

        # Initialize memoization table
        memo = {}
        for i in range(n):
            memo[(1 << i, i)] = (self._get_edge_weight(start_vertex,␣
↪vertices[i]), [start_vertex, vertices[i]])

        # Iterate over subsets of increasing size
        for size in range(2, n + 1):
            for subset in itertools.combinations(range(n), size):
                bits = 0
                for bit in subset:
                    bits |= 1 << bit
                for k in subset:
                    prev_bits = bits & ~(1 << k)
                    res = []
                    for m in subset:
                        if m == k:
                            continue
                        prev_dist, prev_path = memo[(prev_bits, m)]
                        dist = prev_dist + self._get_edge_weight(vertices[m],␣
↪vertices[k])
                        res.append((dist, prev_path + [vertices[k]]))
                    memo[(bits, k)] = min(res)

        # Find the minimum cost to return to the start vertex
        bits = (1 << n) - 1
        res = []
        for k in range(n):
            dist, path = memo[(bits, k)]
            dist += self._get_edge_weight(vertices[k], start_vertex)
            res.append((dist, path + [start_vertex]))

        min_dist, best_path = min(res)
        return (min_dist, best_path)

    def tsp_large_graph(self, start):
        """
        Solve the Travelling Salesman Problem for a large (~1000 node) complete␣
↪graph starting from a specified node.
        No requirement to find the optimal tour. Must run under 0.5 second and␣
↪its solution must no

        Parameters:
        start: The starting node.

        Returns:
        A tuple containing the total distance of the tour and a list of nodes␣
↪representing the tour path.
        """
        nodes = list(self.graph.keys())
        num_nodes = len(nodes)
        adjacency_matrix = np.full((num_nodes, num_nodes), float('inf'))

        node_index = {node: idx for idx, node in enumerate(nodes)}

        for src in self.graph:
            for dest in self.graph[src]:
                adjacency_matrix[node_index[src]][node_index[dest]] = self.
↪graph[src][dest]
```

```
        def calculate_tour_length(tour):
            length =␣
↪sum(adjacency_matrix[node_index[tour[i]]][node_index[tour[i + 1]]] for i in␣
↪range(len(tour) - 1))
            return length

        def nearest_neighbor(start):
            current_node = start
            tour = [current_node]
            unvisited = set(nodes)
            unvisited.remove(current_node)

            while unvisited:
                current_idx = node_index[current_node]
                next_node = min(unvisited, key=lambda node:␣
↪adjacency_matrix[current_idx][node_index[node]])
                tour.append(next_node)
                unvisited.remove(next_node)
                current_node = next_node
            tour.append(start)
            return tour

        def simulated_annealing(initial_tour):
            current_tour = initial_tour
            current_length = calculate_tour_length(current_tour)
            T = 1000
            alpha = 0.995
            while T > 1:
                i, k = sorted(random.sample(range(1, len(current_tour) - 1), 2))
                new_tour = current_tour[:i] + current_tour[i:k+1][::-1] +␣
↪current_tour[k+1:]
                new_length = calculate_tour_length(new_tour)
                if new_length < current_length or random.random() < np.
↪exp((current_length - new_length) / T):
                    current_tour = new_tour
                    current_length = new_length
                T *= alpha

            return current_length, current_tour

        initial_tour = nearest_neighbor(start)
        total_distance, optimized_tour = simulated_annealing(initial_tour)

        return (total_distance, optimized_tour)

    def tsp_medium_graph(self, start):
        """
        Solve the Travelling Salesman Problem for a medium (~300 node) complete␣
↪graph starting from a specified node.
        Expected to perform better than tsp_large_graph. Must run under 1␣
↪second.

        Parameters:
        start: The starting node.

        Returns:
        A tuple containing the total distance of the tour and a list of nodes␣
↪representing the tour path.
        """
        nodes = list(self.graph.keys())
        num_nodes = len(nodes)
        adjacency_matrix = np.full((num_nodes, num_nodes), float('inf'))

        node_index = {node: idx for idx, node in enumerate(nodes)}

        for src in self.graph:
            for dest in self.graph[src]:
                adjacency_matrix[node_index[src]][node_index[dest]] = self.
↪graph[src][dest]

        # Generar un tour inicial utilizando el algoritmo del vecino más cercano
        current_node = start
        tour = [current_node]
        while len(tour) < num_nodes:
            current_idx = node_index[current_node]
            next_node = min((node for node in nodes if node not in tour),␣
↪key=lambda node: adjacency_matrix[current_idx][node_index[node]])
            tour.append(next_node)
            current_node = next_node
        tour.append(start)  # Volver al nodo inicial

        # Función para calcular la longitud del tour
        def calculate_tour_length(tour):
            length = 0
            for i in range(len(tour) - 1):
                length +=␣
↪adjacency_matrix[node_index[tour[i]]][node_index[tour[i + 1]]]
            return length

        # Función para realizar un intercambio 2-opt
        def _2_opt_swap(tour, i, k):
            new_tour = tour[:i] + tour[i:k+1][::-1] + tour[k+1:]
            return new_tour
```

```
    # Algoritmo de Lin-Kernighan simplificado
    best_tour = tour
    best_length = calculate_tour_length(best_tour)
    improved = True

    max_iterations = 5  # Limitar el número de iteraciones
    iteration = 0

    while improved and iteration < max_iterations:
        improved = False
        for i in range(len(best_tour) - 1):
            for k in range(i + 1, len(best_tour)):
                new_tour = _2_opt_swap(best_tour, i, k)
                new_length = calculate_tour_length(new_tour)
                if new_length < best_length:
                    best_tour = new_tour
                    best_length = new_length
                    improved = True
                    break
            if improved:
                break
        iteration += 1

    total_dist = best_length
    best_route = best_tour
    return (total_dist, best_route)
```

1.4 3 - Exercises

1.4.1 3.1 Exercises description

1.4.2 Exercise 1: The Shortest Path Challenge

Develop an algorithm to find the quickest route between two points in a sparse graph containing 10,000 nodes, ensuring it executes in less than `0.5` seconds. If you're unsure about your approach or need optimization advice, the LLM is there to provide insights and strategies.

1.4.3 Exercise 2: Solving the TSP for Small Graphs

Your task is to create a function that solves the Traveling Salesman Problem for small graphs of about 10 nodes, starting at node 0. The solution must be efficient, completing the task in under `1` second. The LLM can serve as a valuable resource for exploring algorithmic solutions and improving your code's efficiency.

1.4.4 Exercise 3: Tackling the TSP for Large Graphs

Address the Traveling Salesman Problem for large graphs with 1,000 nodes. The goal isn't to find the perfect solution but to approach it closely within `0.5` seconds. When faced with the complexity of this challenge, the LLM can offer guidance on heuristic approaches and performance optimization.

1.4.5 Exercise 4: The TSP Medium Graphs Challenge

Focus on medium-sized graphs with 300 nodes and develop an algorithm that performs better than your solution for larger graphs, with a completion time of less than `1.3` seconds. The LLM is ready to assist by sharing insights on algorithmic efficiency and helping you surpass your previous solutions.

1.4.6 3.2 Unit Testing

Unit testing functions are provided to validate your algorithms. Run these without modifications after completing each exercise. They offer immediate feedback, including failure reasons and guidance for improving your solutions. You can test solutions individually, ensuring focused and efficient debugging.

1.4.7 3.3 Generate Graph Function

The function outlined below can assist you in experimenting, debugging, and testing your code while you work on the algorithms. If your algorithm fails a test case, you will be given the call to this function with the appropriate arguments needed to replicate the graph that caused your algorithm to fail. Additionally, the reason for the failure will be provided, whether it was due to exceeding the time limit or not achieving an optimal or near-optimal distance. Feel free to ask the LLM to explain how this function works.

[23]:
```
def generate_graph(nodes, edges=None, complete=False, weight_bounds=(1,600),␣
↪seed=None):
    """
    Generates a graph with specified parameters, allowing for both complete and␣
↪incomplete graphs.

    This function creates a graph with a specified number of nodes and edges,␣
↪with options for creating a complete graph, and for specifying the weight␣
↪bounds of the edges. It uses the Graph_Advanced class to create and␣
↪manipulate the graph.

    Parameters:
    - nodes (int): The number of nodes in the graph. Must be a positive integer.
    - edges (int, optional): The number of edges to add for each node in the␣
↪graph. This parameter is ignored if `complete` is set to True. Defaults to␣
↪None.
    - complete (bool, optional): If set to True, generates a complete graph␣
↪where every pair of distinct vertices is connected by a unique edge.␣
↪Defaults to False.
    - weight_bounds (tuple, optional): A tuple specifying the lower and upper␣
↪bounds (inclusive) for the random weights of the edges. Defaults to (1, 600).
```

```
    - seed (int, optional): A seed for the random number generator to ensure␣
↪reproducibility. Defaults to None.

    Raises:
    - ValueError: If `edges` is not None and `complete` is set to True, since a␣
↪complete graph does not require specifying the number of edges.

    Returns:
    - Graph_Advanced: An instance of the Graph_Advanced class representing the␣
↪generated graph, with vertices labeled as integers starting from 0.

    Examples:
    - Generating a complete graph with 5 nodes:
        generate_graph(5, complete=True)

    - Generating an incomplete graph with 5 nodes and 2 edges per node:
        generate_graph(5, edges=2)

    Note:
    - The function assumes the existence of a Graph_Advanced class with methods␣
↪for adding vertices (`add_vertex`) and edges (`add_edge`), as well as a␣
↪method for getting adjacent vertices (`get_adjacent_vertices`).
    """
    random.seed(seed)
    graph = Graph_Advanced()
    if edges is not None and complete:
        raise ValueError("edges must be None if complete is set to True")
    if not complete and edges > nodes:
        raise ValueError("number of edges must be less than number of nodes")


    for i in range(nodes):
        graph.add_vertex(i)
    if complete:
        for i in range(nodes):
            for j in range(i+1,nodes):
                weight = random.randint(weight_bounds[0], weight_bounds[1])
                graph.add_edge(i,j,weight)
    else:
        for i in range(nodes):
            for _ in range(edges):
                j = random.randint(0, nodes - 1)
                while (j == i or j in graph.get_adjacent_vertices(i)) and␣
↪len(graph.get_adjacent_vertices(i)) < nodes - 1:  # Ensure the edge is not a␣
↪loop or a duplicate
                    j = random.randint(0, nodes - 1)
                weight = random.randint(weight_bounds[0], weight_bounds[1])
                if len(graph.graph[i]) < edges and len(graph.graph[j]) < edges:
                    graph.add_edge(i, j, weight)
    return graph
```

1.5 4 - Playground

This section is your space to experiment with and test your methods. To measure the execution time of your code, you can use the `%%timeit` magic method. Remember, `%%timeit` should be placed at the **beginning** of a code block, even before any comments marked by `#`. It's important to note that this magic method doesn't save the output of the function it measures. You're encouraged to create as many new cells as needed for testing, but keep in mind that only the code within the `Graph_Advanced` class will be considered during grading. Here's an example of how to use `%%timeit` to assess the execution time of functions.

[24]:
```
generate_graph(nodes = 1000, complete = True, seed = 42)
#generate_graph(nodes = 300, complete = True, seed = 48)
#generate_graph(nodes = 100, edges = 100, seed = 42)
```

[24]:
```
<__main__.Graph_Advanced at 0x7dfb02128310>
```

[25]:
```
print(generate_graph(nodes = 1000, complete = True, seed = 42))
#print(generate_graph(nodes = 300, complete = True, seed = 48))
#print(generate_graph(nodes = 100, edges = 100, seed = 42))
```

```
IOPub data rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_data_rate_limit`.

Current values:
ServerApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
ServerApp.rate_limit_window=3.0 (secs)
```

[20]:
```
%%timeit
generate_graph(nodes = 1000, complete = True, seed = 42)
#generate_graph(nodes = 300, complete = True, seed = 48)
#generate_graph(nodes = 100, edges = 100, seed = 42)
```

```
666 ms ± 68.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Feel free to add as many cells as you want! You can do so by clicking in the + button in the left upper corner of this notebook.

[26]:
```
graph_example = generate_graph(nodes = 300, complete = True, seed = 42)
```

1.6 5 - Test Your Solutions

After implementing your methods in the previous class, you can run the code blocks below to test each of your developed methods! If you see the message "All tests passed!" for every unittest, your assignment is likely ready for submission. If not, you'll be able to see which cases your code didn't pass and understand why. **Remember, the unittests here cover only a few example cases due to the constraints of this environment. The grading process will include more extensive testing. It's possible to pass all the tests here but still fail some during the autograding.** However, if that happens, don't worry—you will receive feedback on why you failed, which will allow you to adjust your functions and resubmit the assignment. You're welcome to submit your assignment multiple times as needed.

IMPORTANT NOTE: Please ensure you shut down any other live kernels (if you have another notebook open within this environment), as they may affect the execution time of your functions.

1.6.1 5.1 Test Exercise 1 (method Graph_Advanced.shortest_path)

Run the code below to test the `shortest_path` method on sparsely connected graphs with 10,000 nodes. The requirements for passing this exercise are:

- The algorithm must complete its run in under `0.5` second for each graph.
- It must accurately find the shortest path.

```
[88]: unittests.test_shortest_path(Graph_Advanced)
```

```
All tests passed!
```

1.6.2 5.2 Test Exercise 2 (method Graph_Advanced.tsp_small_graph)

Run the code below to test the `tsp_small_graph` on complete (fully connected) graphs with 10 nodes. The requirements for passing this exercise are:

- The algorithm must complete its run in under `1` second for each graph.
- It must fund the optimal solution, starting at node 0.

```
[10]: unittests.test_tsp_small_graph(Graph_Advanced)
```

```
All tests passed!
```

1.6.3 5.3 Test Exercise 3 (method Graph_Advanced.tsp_large_graph)

Run the code below to test the `tsp_large_graph` on complete (fully connected) graphs with 1000 nodes. The requirements for passing this exercise are:

- The algorithm must complete its run in under `0.5` second for each graph.
- It must find the good solution (less than a specified value, depending on the graph).

```
[27]: unittests.test_tsp_large_graph(Graph_Advanced)
```

```
Failed test case: Failed to find the optimal solution for a path starting
in node 0. To replicate the graph, you may run generate_graph(nodes = 1000,
complete = True, seed = 42), real 47740.0.
Expected: 4367
Got: 47740.0
Failed test case: Failed to find the optimal solution for a path starting
in node 0. To replicate the graph, you may run generate_graph(nodes = 1000,
complete = True, seed = 43), real 54435.0.
Expected: 4774
Got: 54435.0
Failed test case: Failed to find the optimal solution for a path starting
in node 0. To replicate the graph, you may run generate_graph(nodes = 1000,
complete = True, seed = 44), real 45858.0.
Expected: 5217
Got: 45858.0
Failed test case: Failed to find the optimal solution for a path starting
in node 0. To replicate the graph, you may run generate_graph(nodes = 1000,
complete = True, seed = 45), real 50255.0.
Expected: 4357
Got: 50255.0
Failed test case: Failed to find the optimal solution for a path starting
in node 0. To replicate the graph, you may run generate_graph(nodes = 1000,
complete = True, seed = 46), real 46041.0.
Expected: 4613
Got: 46041.0
```

1.6.4 5.4 Test Exercise 4 (method Graph_Advanced.tsp_medium_graph)

Run the code below to test the `tsp_medium_graph` on complete (fully connected) graphs with 300 nodes. The requirements for passing this exercise are:

- The algorithm must complete its run in under `1.3` seconds for each graph.
- It must find the good solution (less than a specified value, depending on the graph).

[26]:
```
unittests.test_tsp_medium_graph(Graph_Advanced)
```

```
All tests passed!
```

Congratulations! You completed the first assignment of the Specialization!