CSE 331 **Semester FS18**

Honors Project 02

Authors of this project are Daniel Webb and Caitlin Brown

Assignment Overview

In this project we will be using an **adjacency list** that uses a list as the underlying container. The goal of this project is to encourage you to use the concepts taught throughout the semester to solve a unique and complex problem.

For this assignment you will be implementing a basic Graph ADT. Test cases will be provided for you to test your code, along with a skeleton file for you to start with where a Graph, Vertex, Path, and Edge class have already been declared for you.

Assignment Deliverables

Be sure to submit the following files:

- **Graph.py**
- **Readme.txt** which includes:
    1. Your name, section, and title of assignment.
    2. Feedback on the project and how long it took to complete.
    3. Any external resources you used to help.

Assignment Specifications

The Edge class is fully implemented and provided for you. **Do not modify this class.**

We have provided the __init__, __eq__, methods in the Graph class, **do not modify these methods.** Your task will be to complete the methods listed below in the Graph, Vertex, and Path classes. Make sure that you are adhering to the time and space complexity requirements. **Do not modify function signatures in any way.**

Edge Class:

This class is provided for you, DO NOT modify it. An edge object represents an edge in the graph. It connects a source (Vertex object) with a destination (Vertex id).

Path Class:

self.vertices represents the vertices in the path in a specific order.

This class keeps track of a path through the graph. This is useful in graphs because it allows for a meaningful representation of what a path is. i.e. A route in google maps is represented by a path through a graph.

- add_vertex(self, vertex):
    - Add a Vertex id to the path.
    - Return None
    - O(1) time complexity
- remove_vertex(self):
    - Remove the most recently added Vertex id from the path.
    - Return None
    - O(1) time complexity
- last_vertex(self):
    - Return the last Vertex id added to the path
    - If path is empty return None
    - O(1) time complexity
- is_empty(self):
    - Check if the path is empty.
    - Return Boolean
    - O(1) time complexity

Vertex Class:

- This class represents a vertex in the Graph.
    - self.edges is a list of outgoing edges from the vertex
    - self.ID is the id of the vertex
    - Self.pred_vertex is set by your implementation of the Bellman Ford algorithm as the preceding vertex on the shortest path.
    - Self.distance is also used in your implementation of Bellman Ford to calculate the shortest path length.
- add_edge(self, destination):
    - Add an edge to the Vertex given the id of the destination Vertex.
    - Return None
    - O(1) time complexity
- get_edge(self, destination):
    - Returns the Edge that goes to a specified destination node.
    - If the edge is not found, return None
    - O(n) time complexity
- get_edges(self):

- Returns a list of all of the edges.
- O(1) time complexity

Graph Class:

- An abstract class that represents a directed graph
  - self.adj_list represents the adjacency list storing the graph. Structure: {vertex_id: Vertex()}
- get_vertex(self, id):
  - Returns the vertex with the specified id.
  - If the vertex is not found, return None
  - O(1) time complexity
- construct_edge(self, (source, destination)):
  - Add the given edge to the graph.
  - return data in the following format: [source, destination]
  - Uses the dictionary self.adj_list to store vertices' IDs as keys and their objects as values.
  - O(E) time complexity, E is the number of edges to insert.
- bellman_ford(self):
  - Perform the Bellman-Ford shortest path algorithm to update each vertex's predecessor.
  - O(VE), V is the number of vertices, E is the number of edges
- get_shortest_path(self, startID, endID):
  - Call Bellman Ford to update graph predecessors.
  - Return the shortest path between startID and endID as a Path object.
  - Path should be empty in there is no shortest path.
  - Raise GraphError if the Vertices do not exist in the graph.
  - O(V+E), V is the number of vertices, E is the number of edges

Assignment Notes

- You are required to add and complete docstrings for each function that you complete.
- You are provided with skeleton code for the classes and you must complete each empty function. You may use more functions if you'd like, but you must complete the ones given to you. If you do choose to make more functions, you are required to complete docstrings for those as well.
- Make sure that you are adhering to all specifications for the functions, including time complexity.