Wesley Kwan

CS 2400.02

Project 3

May 13, 2020

Section 1:

Interface BasicGraphInterface<T>

- **Method Summary**

Modifier and Type	Method and Description
boolean	addEdge(T begin, T end) Adds an unweighted edge between two given distinct vertices that are currently in this graph.
boolean	addEdge(T begin, T end, double edgeWeight) Adds a weighted edge between two given distinct vertices that are currently in this graph.
boolean	addVertex(T vertexLabel) Adds a given vertex to this graph.
void	clear() Removes all vertices and edges from this graph resulting in an empty graph.
int	getNumberOfEdges() Gets the number of edges in this graph.
int	getNumberOfVertices() Gets the number of vertices in this graph.
boolean	hasEdge(T begin, T end) Sees whether an edge exists between two given vertices.
boolean	isEmpty() Sees whether this graph is empty.
boolean	removeEdge(T begin, T end) Removes an unweighted edge between two given distinct vertices that are currently in this graph.

- **Method Detail**
  - **addVertex**

    boolean addVertex(T vertexLabel)
    Adds a given vertex to this graph.
    Parameters:
    vertexLabel - An object that labels the new vertex and is distinct from the labels of current vertices.
    Returns:
    True if the vertex is added, or false if not.
  - **addEdge**

    boolean addEdge(T begin, T end, double edgeWeight)
    Adds a weighted edge between two given distinct vertices that are currently in this graph. The desired edge must not already be in the graph. In a directed graph, the edge points toward the second vertex given.
    Parameters:
    begin - An object that labels the origin vertex of the edge.
    end - An object, distinct from begin, that labels the end vertex of the edge.
    edgeWeight - The real value of the edge's weight.
    Returns:
    True if the edge is added, or false if not.
  - **addEdge**

    boolean addEdge(T begin, T end)
    Adds an unweighted edge between two given distinct vertices that are currently in this graph. The desired edge must not already be in the graph. In a directed graph, the edge points toward the second vertex given.
    Parameters:
    begin - An object that labels the origin vertex of the edge.
    end - An object, distinct from begin, that labels the end vertex of the edge.
    Returns:
    True if the edge is added, or false if not.
  - **removeEdge**

    boolean removeEdge(T begin, T end)
    Removes an unweighted edge between two given distinct vertices that are currently in this graph. In a directed graph, the edge points toward the second vertex given.
    Parameters:
    begin - An object that labels the origin vertex of the edge.
    end - An object, distinct from begin, that labels the end vertex of the edge.
    Returns:
    True if the edge is removed, or false if not.

- **hasEdge**
  boolean hasEdge(T begin, T end)
  Sees whether an edge exists between two given vertices.
  Parameters:
  begin - An object that labels the origin vertex of the edge.
  end - An object that labels the end vertex of the edge.
  Returns:
  True if an edge exists.
- **isEmpty**
  boolean isEmpty()
  Sees whether this graph is empty.
  Returns:
  True if the graph is empty.
- **getNumberOfVertices**
  int getNumberOfVertices()
  Gets the number of vertices in this graph.
  Returns:
  The number of vertices in the graph.
- **getNumberOfEdges**
  int getNumberOfEdges()
  Gets the number of edges in this graph.
  Returns:
  The number of edges in the graph.
- **clear**
  void clear()
  Removes all vertices and edges from this graph resulting in an empty graph.

Interface GraphAlgorithmsInterface<T>

- **Method Summary**

Modifier and Type	Method and Description
java.util.Queue<T>	getBreadthFirstTraversal(T origin) Performs a breadth-first traversal of this graph.
double	getCheapestPath(T begin, T end, java.util.Stack<T> path) Finds the least-cost path between two given vertices in this graph.
java.util.Queue<T>	getDepthFirstTraversal(T origin) Performs a depth-first traversal of this graph.
int	getShortestPath(T begin, T end, java.util.Stack<T> path) Finds the shortest-length path between two given vertices in this graph.
java.util.Stack<T>	getTopologicalOrder() Performs a topological sort of the vertices in this graph without cycles.

- **Method Detail**
  - **getBreadthFirstTraversal**
    java.util.Queue<T> getBreadthFirstTraversal(T origin)
    Performs a breadth-first traversal of this graph.
    Parameters:
    origin - An object that labels the origin vertex of the traversal.
    Returns:
    A queue of labels of the vertices in the traversal, with the label of the origin vertex at the queue's front.
  - **getDepthFirstTraversal**
    java.util.Queue<T> getDepthFirstTraversal(T origin)
    Performs a depth-first traversal of this graph.
    Parameters:
    origin - An object that labels the origin vertex of the traversal.
    Returns:
    A queue of labels of the vertices in the traversal, with the label of the origin vertex at the queue's front.
  - **getTopologicalOrder**
    java.util.Stack<T> getTopologicalOrder()
    Performs a topological sort of the vertices in this graph without cycles.
    Returns:
    A stack of vertex labels in topological order, beginning with the stack's top.

- **getShortestPath**
  int getShortestPath(T begin, T end, java.util.Stack<T> path)
  Finds the shortest-length path between two given vertices in this graph.
  Parameters:
  begin - An object that labels the path's origin vertex.
  end - An object that labels the path's destination vertex.
  path - A stack of labels that is empty initially; at the completion of the method, this stack contains the labels of the vertices along the shortest path; the label of the origin vertex is at the top, and the label of the destination vertex is at the bottom
  Returns:
  The length of the shortest path.
- **getCheapestPath**
  double getCheapestPath(T begin, T end, java.util.Stack<T> path)
  Finds the least-cost path between two given vertices in this graph.
  Parameters:
  begin - An object that labels the path's origin vertex.
  end - An object that labels the path's destination vertex.
  path - A stack of labels that is empty initially; at the completion of the method, this stack contains the labels of the vertices along the cheapest path; the label of the origin vertex is at the top, and the label of the destination vertex is at the bottom
  Returns:
  The cost of the cheapest path.

Interface GraphInterface<T>

- **Method Summary**
  - **Methods inherited from interface BasicGraphInterface**
    addEdge, addEdge, addVertex, clear, getNumberOfEdges, getNumberOfVertices, hasEdge, isEmpty, removeEdge
  - **Methods inherited from interface GraphAlgorithmsInterface**
    getBreadthFirstTraversal, getCheapestPath, getDepthFirstTraversal, getShortestPath, getTopologicalOrder

Interface VertexInterface<T>

- **Method Summary**

Modifier and Type	Method and Description
boolean	connect(VertexInterface<T> endVertex) Connects this vertex and a given vertex with an unweighted edge.
boolean	connect(VertexInterface<T> endVertex, double edgeWeight) Connects this vertex and a given vertex with a weighted edge.
boolean	disconnect(VertexInterface<T> endVertex) Disconnects this vertex and a given vertex with an unweighted edge.
double	getCost() Gets the recorded cost of the path to this vertex.
T	getLabel() Gets this vertex's label.
java.util.Iterator<VertexInterface<T>>	getNeighborIterator() Creates an iterator of this vertex's neighbors by following all edges that begin at this vertex.
VertexInterface<T>	getPredecessor() Gets the recorded predecessor of this vertex.
VertexInterface<T>	getUnvisitedNeighbor() Gets an unvisited neighbor, if any, of this vertex.
java.util.Iterator<java.lang.Double>	getWeightIterator() Creates an iterator of the weights of the edges to this vertex's neighbors.
boolean	hasNeighbor() Sees whether this vertex has at least one neighbor.
boolean	hasPredecessor() Sees whether a predecessor was recorded for this vertex.
boolean	isVisited() Sees whether the vertex is marked as visited.
void	setCost(double newCost) Records the cost of a path to this vertex.

void	setPredecessor(VertexInterface<T> predecessor) Records the previous vertex on a path to this vertex.
void	unvisit() Removes this vertex's visited mark.
void	visit() Marks this vertex as visited.

- **Method Detail**
  - **getLabel**
    T getLabel()
    Gets this vertex's label.
    Returns:
    The object that labels the vertex.
  - **visit**
    void visit()
    Marks this vertex as visited.
  - **unvisit**
    void unvisit()
    Removes this vertex's visited mark.
  - **isVisited**
    boolean isVisited()
    Sees whether the vertex is marked as visited.
    Returns:
    True if the vertex is visited.
  - **connect**
    boolean connect(VertexInterface<T> endVertex, double edgeWeight)
    Connects this vertex and a given vertex with a weighted edge. The two vertices cannot be the same, and must not already have this edge between them. In a directed graph, the edge points toward the given vertex.
    Parameters:
    endVertex - A vertex in the graph that ends the edge.
    edgeWeight - A real-valued edge weight, if any.
    Returns:
    True if the edge is added, or false if not.
  - **connect**
    boolean connect(VertexInterface<T> endVertex)
    Connects this vertex and a given vertex with an unweighted edge. The two vertices cannot be the same, and must not already have this edge between them. In a directed graph, the edge points toward the given vertex.
    Parameters:
    endVertex - A vertex in the graph that ends the edge.
    Returns:
    True if the edge is added, or false if not.
  - **disconnect**
    boolean disconnect(VertexInterface<T> endVertex)
    Disconnects this vertex and a given vertex with an unweighted edge. The two vertices cannot be the same. In a directed graph, the edge points toward the given vertex.
    Parameters:
    endVertex - A vertex in the graph that ends the edge.
    Returns:
    True if the edge is removed, or false if not.
  - **getNeighborIterator**
    java.util.Iterator<VertexInterface<T>> getNeighborIterator()
    Creates an iterator of this vertex's neighbors by following all edges that begin at this vertex.
    Returns:
    An iterator of the neighboring vertices of this vertex.
  - **getWeightIterator**
    java.util.Iterator<java.lang.Double> getWeightIterator()
    Creates an iterator of the weights of the edges to this vertex's neighbors.
    Returns:
    An iterator of edge weights for edges to neighbors of this vertex.
  - **hasNeighbor**
    boolean hasNeighbor()
    Sees whether this vertex has at least one neighbor.
    Returns:
    True if the vertex has a neighbor.

- **getUnvisitedNeighbor**
  VertexInterface<T> getUnvisitedNeighbor()
  Gets an unvisited neighbor, if any, of this vertex.
  Returns:
  Either a vertex that is an unvisited neighbor or null if no such neighbor exists.
- **setPredecessor**
  void setPredecessor(VertexInterface<T> predecessor)
  Records the previous vertex on a path to this vertex.
  Parameters:
  predecessor - The vertex previous to this one along a path.
- **getPredecessor**
  VertexInterface<T> getPredecessor()
  Gets the recorded predecessor of this vertex.
  Returns:
  Either this vertex's predecessor or null if no predecessor was recorded.
- **hasPredecessor**
  boolean hasPredecessor()
  Sees whether a predecessor was recorded for this vertex.
  Returns:
  True if a predecessor was recorded.
- **setCost**
  void setCost(double newCost)
  Records the cost of a path to this vertex.
  Parameters:
  newCost - The cost of the path.
- **getCost**
  double getCost()
  Gets the recorded cost of the path to this vertex.
  Returns:
  The cost of the path.

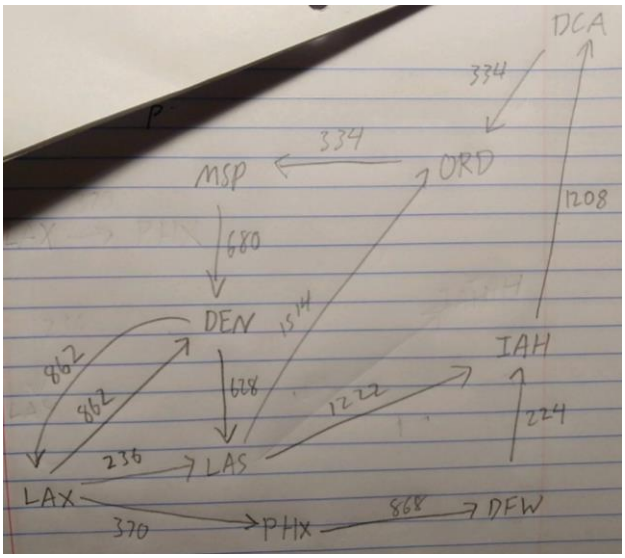
Description:
Instead of writing my own queue or stack interface and implementation, I use ones provided by Java's util library. Also, Java's Iterator class is used in VertexInterface, Vertex, and DirectedGraph. Iterator is used in Vertex to create the inner NeighborIterator and WeightIterator classes, which are used in some methods in Vertex and getCheapestPath in DirectedGraph. Vertex class uses Java's List interface and LinkedList class to store and access the edges of a vertex. These edges are instances of the inner class Edge. DirectedGraph uses Java's Map interface (Java's replacement of the Dictionary ADT) and HashMap class to store and access the vertices (Vertex instances) in the graph. Many of the methods in DirectedGraph call on the instance methods of Vertex to accomplish their purpose. Java's Queue and Stack are primarily used in DirectedGraph, specifically for the implementations of graph algorithms. getCheapestPath uses Java's Queue interface and PriorityQueue class to manage the vertex entries. These entries are instances of the inner class EntryPQ that implements Java's Comparable interface. getCheapestPath uses Java's Stack class to store the cheapest path found by the algorithm. AirportApp reads in two files. It first reads airports.csv. It adds vertices into a graph using the airport codes as labels, and it stores the data in a dictionary with airport codes as the keys and their respective airport names as the values. Then AirportApp reads distances.csv and uses the data to create the edges in the graph. Afterward, AirportApp prompts the user for a command.

Section 2:

LAX	Los Angeles (CA) - International
PHX	Phoenix (AZ) - Sky Harbor International
DFW	Dallas/Ft. Worth (TX) - Dallas/Fort Worth International
DEN	Denver (CO) - Denver International Airport
LAS	Las Vegas (NV) - McCarran International Airport
IAH	Houston (TX) - George Bush Intercontinental Airport
MSP	Minneapolis (MN) - St. Paul International Airport
ORD	Chicago (IL) - O'Hare International Airport
DCA	Washington DC - Ronald Reagan National

LAX	PHX	370
LAX	LAS	236
LAX	DEN	862
DEN	LAS	628
MSP	DEN	680
PHX	DFW	868
LAS	IAH	1222
LAS	ORD	1514
ORD	MSP	334
DFW	IAH	224
IAH	DCA	1208
DCA	ORD	334
DEN	LAX	862

Before AirportApp prompts the user for a command, two data structures are formed: a dictionary and a graph. The dictionary stores the data from airport.csv (left table) and should be represented by the table with the left column as the keys and right column as the values. The graph is generated with the data from distances.csv (right table) and should be represented by the diagram in the picture below:



By using the two representations of the data structures and the user commands provided by AirportApp, I can determine whether the program and methods work properly.

Case 1: Check if the dictionary correctly maps key and value (airport code and name)
Case 2: Check if the graph matches the diagram
Case 3: Check if the graph is updated after removing a connection
Case 4: Check if the graph is updated after adding a connection

Altogether, these cases cover the interconnected adding and removing methods implemented in Vertex and DirectedGraph, which are used to build and manipulate the graph. getCheapestPath is covered in the process.

Case 1: Check Dictionary
Output:
Command? Q
Airport code: LAX
Los Angeles (CA) - International
Command? Q
Airport code: LAP
Airport code does not exist.

Given an existing airport code (in airports.csv), the program always prints the respective airport name. A warning is printed when an unrecognized airport code is inputted. Therefore, it is safe to assume that the dictionary is correctly mapped.

Case 2: Check Graph
Whether the graph is built correctly can be determined by calling getCheapestPath method on opposite ends of the graph and comparing the results with the diagram.

Output:
Command? D
Airport codes: LAX DCA
The minimum distance between Los Angeles (CA) - International and Washington DC - Ronald Reagan National is 2666 through the route:
Los Angeles (CA) - International
Las Vegas (NV) - McCarran International Airport
Houston (TX) - George Bush Intercontinental Airport
Washington DC - Ronald Reagan National

After manually tracing the cheapest path and path cost using the diagram, I can confirm that the getCheapestPath found the cheapest path. Since getCheapestPath uses the graph built by the program, the program must have at least correctly built the respective portion of the graph.

Output:
Command? D
Airport codes: DCA LAX
The minimum distance between Washington DC - Ronald Reagan National and Los Angeles (CA) - International is 2210 through the route:
Washington DC - Ronald Reagan National
Chicago (IL) - O'Hare International Airport
Minneapolis (MN) - St. Paul International Airport
Denver (CO) - Denver International Airport
Los Angeles (CA) - International
Command? D

Airport codes: DFW MSP
The minimum distance between Dallas/Ft. Worth (TX) - Dallas/Fort Worth International and Minneapolis (MN) - St. Paul International Airport is 2100 through the route:
Dallas/Ft. Worth (TX) - Dallas/Fort Worth International
Houston (TX) - George Bush Intercontinental Airport
Washington DC - Ronald Reagan National
Chicago (IL) - O'Hare International Airport
Minneapolis (MN) - St. Paul International Airport
Command? D
Airport codes: MSP DFW
The minimum distance between Minneapolis (MN) - St. Paul International Airport and Dallas/Ft. Worth (TX) - Dallas/Fort Worth International is 2780 through the route:
Minneapolis (MN) - St. Paul International Airport
Denver (CO) - Denver International Airport
Los Angeles (CA) - International
Phoenix (AZ) - Sky Harbor International
Dallas/Ft. Worth (TX) - Dallas/Fort Worth International

After repeating this process with the other corners of the graph and always receiving the correct path, it is likely that the getCheapestPath method is written correctly. Likewise, if getCheapestPath always produces the same path that I have traced using the diagram, then the diagram is representative of the graph, and thus the graph is built correctly by the program. This means the methods that add vertices and edges, used to build the graph at the start of the program, in Vertex and Directed graph are written correctly.

Case 3: Removing a Connection
Output:
Command? D
Airport codes: LAX DCA
The minimum distance between Los Angeles (CA) - International and Washington DC - Ronald Reagan National is 2666 through the route:
Los Angeles (CA) - International
Las Vegas (NV) - McCarran International Airport
Houston (TX) - George Bush Intercontinental Airport
Washington DC - Ronald Reagan National
Command? R
Airport codes: LAX LAS
The connection from Los Angeles (CA) - International and Las Vegas (NV) - McCarran International Airport removed.
Command? D
Airport codes: LAX DCA
The minimum distance between Los Angeles (CA) - International and Washington DC - Ronald Reagan National is 2670 through the route:

Los Angeles (CA) - International
Phoenix (AZ) - Sky Harbor International
Dallas/Ft. Worth (TX) - Dallas/Fort Worth International
Houston (TX) - George Bush Intercontinental Airport
Washington DC - Ronald Reagan National

A different path is produced when getCheapestPath is called after removing the connection between LAX and LAS compared to the path before the removal. This new path is the cheapest is in line with the diagram if said connection were removed. Therefore, the connection was removed in the program's graph. This means the methods that remove edges, used to remove a connection, in Vertex and DirectedGraph are written correctly.

Case 4: Adding a Connection
If the removed connection between LAX and LAS were inserted back in, then the path produced by getCheapestPath should match the path before the path was ever removed.

Output:
Command? I
Airport codes and distance: LAX LAS 236
You have inserted a connection from Los Angeles (CA) - International to Las Vegas (NV) - McCarran International Airport with a distance of 236.
Command? D
Airport codes: LAX DCA
The minimum distance between Los Angeles (CA) - International and Washington DC - Ronald Reagan National is 2666 through the route:
Los Angeles (CA) - International
Las Vegas (NV) - McCarran International Airport
Houston (TX) - George Bush Intercontinental Airport
Washington DC - Ronald Reagan National

After inserting the connection between LAX and LAS with the same weight as the removed connection, getCheapestPath produced the same path as the original. Therefore, the connection was inserted in the program's graph. This further confirms the methods that add edges to the graph are written correctly.

Section 3:
From this project I learned that data structures are versatile. I was surprised that a graph could be made of a dictionary and lists, connected by pointers and addresses. Through the process of completing the project, I have gained a better understanding of how different classes are able to interact with one another.